

架构探险

轻量级微服务架构

下册

黄勇 著

电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

本书将重点关注微服务基础设施方面，其中大部分内容涉及微服务运维相关技术。全书以实践的角度进行编写，读者首先将学习轻量级微服务架构的全景视图，随后的各个章节将围绕微服务的日志、监控、通信、解耦、测试、配置六大方面进行展开。读者可亲自动手，从零开始搭建轻量级微服务架构，充分享受架构探险的乐趣。

本书适合对微服务实践感兴趣，以及想成为微服务架构师的人员阅读。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

架构探险：轻量级微服务架构. 下册 / 黄勇著. —北京：电子工业出版社，2017.9
ISBN 978-7-121-32447-5

I. ①架… II. ①黄… III. ①互联网络—网络服务器—研究 IV. ①TP368.5

中国版本图书馆 CIP 数据核字（2017）第 191711 号

责任编辑：陈晓猛

印 刷：

装 订：

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱

邮编：100036

开 本：787×980 1/16 印张：21.25

字数：408 千字

版 次：2017 年 9 月第 1 版

印 次：2017 年 9 月第 1 次印刷

定 价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

推荐序

微服务，我们如何与你相处

微服务来了，有了“服务”这两个字，这注定又是个一说就明白、一举例就糊涂、一讨论就吵架的概念。微服务的出现有其必然的商业背景和架构哲学，如何更好地认识微服务的内涵、如臂使指地应用微服务架构，还是有着很多挑战的，这也许就是本书被命名为“架构探险”的原因。

企业数字化转型驱动架构升级

互联网经济深刻改变了我们身边的商业环境，消费者的生活方式日益数字化，人们可以在任何时间、任何地点利用线上、线下渠道体验无缝购物，运用社交媒体表达自我，企业也在运用多种技术手段，发挥数字化潜力，改善客户联系，促进企业业务模式的转型。Gartner 认为，数字化就是把人、事、物和商业联系起来，建立新的商业模式。未来的企业都将是 IT 企业，IT 将从后台走向前台，从 ERP、CRM 等内部流程优化为主的业务，逐步转向内外兼修的模式，从而实现商业创新。

这一变化要求 IT 架构更加灵活地与上下游企业协作，更加快速地响应客户的个性化需求，更加弹性地应对无时不在的客户请求并提供良好的客户体验，同时云计算、大数据等技术的出现也为上述改变提供了新的技术选择，我们正面临 B/S 多层架构出现后新的一次架构升级，而微服务架构就是在这个架构升级过程中应运而生的。

分而治之的哲学是微服务的理论基础

把大的问题分解为容易解决的小问题，找到小问题的解决办法，再来解决大问题，这就是分而治之的哲学。正如万事万物由分子、原子组成一样，软件也可以分解为基本单元，以这样的基本单元进行开发、测试、维护，是解决大规模系统建设的思路。分而治之首先要解决如何分的问题，企业软件的分法应该是以业务驱动的，而不是以技术驱动的，也就是分解为独立的

业务逻辑，而这样的不可再分的业务逻辑就是微服务。

凡事有一利必有一弊，细分为微服务后，势必带来部署、测试、信息集成难度的提高，分而治之除了“分”，还需要“治”。传统恐龙型 ERP 是一个面向组织的软件，完备、复杂、响应变化慢，适合业务稳定的情况，而在数字化时代，客户个性化的要求让我们从这种面向组织的软件逐渐演变为面向个体的软件。例如，从前的 EHR 软件是为人力资源部门服务的，整体开发、整体实施，而现在我们会从个体的角度规划软件，可以先从招聘专员开始做一个面试管理的流程，逐步推出新的流程，完善现有的流程。这些面向个体的流程就是微应用，企业应用将由无数个微应用组成。微服务则是一个技术概念，能更好地解决微应用的技术实现问题，是一个事物的不同侧面，所谓“横看成岭侧成峰，远近高低各不同”，微服务和微应用是事物的一体两面。正因为微服务实际就是一个业务逻辑，因此做好微服务需要从微应用的维度考虑，将分解开的逻辑形成一个整体，要从多渠道接入、客户体验、数据管理、应用交付、运维全方位的视角考虑，这就是分而治之之中实现“治”的体验，也是微服务架构需要解决的问题。

站在 SOA 的肩膀上践行微服务

微服务是一个新概念，但这绝不是一个全新架构，更不是一个包治百病的架构。由于有服务二字，很容易让人联想到面向服务架构（SOA），其实微服务架构属于应用技术架构，和以 B/S 为代表的三层架构相对应，强调将巨石型应用拆分为由微服务组成的应用，在数据上也视情况从集中的存储拆解为更小的存储单元。而 SOA 属于企业架构的范畴，从企业架构出发把业务分解为不同领域的服务，不同物理系统提供不同服务，注重系统之间通过服务互联互通的规范，对服务如何实现并不关注。因此，面向服务架构的服务应该是一个业务意义的服务，而微服务是系统中的技术服务，更关注服务的实现，虽然提供了业务意义的服务，但是不能混为一谈。微服务使用也不是无限度的，事实上由于数据一致性等问题的限制，不能无限度拆分微服务，因此可以把微服务分为系统对外提供的远程服务、系统内部的远程服务和系统内部的本地服务，显式声明、明确职责。事实上，在企业架构上使用 SOA 支撑业务，而在应用技术架构上使用微服务架构，是一个合适的选择。

黄柳青博士是我和黄勇共同的导师，他在 2004 年所著的《软件的涅槃》一书中指出：“互联网时代的企业应用定义，正发生革命性的变化……横向的部门互动、实时的企业间互动、多样的交互渠道、灵活的业务规则，使得原有意义上的独立应用不复存在……对软件设计者来说，能直观地分割并具有最小内部耦合的软件结构是简约之美……美的软件是软件企业与软件开发者的终极目标”，那时候他把这种全新的软件生产模式称为“面向构件”。回头看来，微服务正是“面向构件”在数字化时代的解读，用微服务架构实现软件之美，加速企业数字化转型。

——焦烈焱，普元 CTO

专家推荐

（排名不分先后）

SOA 从企业级应用到互联网领域火了很多年，曾经是我招聘架构师的必考题目之一，但 SOA 在大型系统的落地从来都是高难度动作，令许多架构师欲仙欲死。如今又兴起了微服务架构，要把 SOA 进行到底，实现彻底的服务化，从此世间再无系统切分，只有微服务小而美好。那么到底如何实现微服务呢？黄老师这本书教我们轻松上手，一步步把理想变成现实，体现出多年实战派的底蕴，是一本不可多得的武功秘籍。

—— 史海峰，饿了么北京研发中心总经理

近年来，微服务俨然成为行业内广受关注的热点。不论是微服务的价值，还是微服务的阻碍，都是行业在架构技术选型中最为关心的前提。除此之外，技术的践行流程，对现有组织架构、软件模式的影响，都是决策者不敢忽视的要素。我很庆幸看到，国内能诞生这本微服务领域的巨著。本书从架构发展史的角度，阐述了微服务兴起的客观性与必然性；从技术的角度，深入分析了践行微服务的种种要点；更从实践的角度，通过案例事无巨细地帮助读者去体会、理解、掌握微服务。实属呕心沥血之作，极力推荐大家阅读。

—— 孙宏亮，DaoCloud 技术合伙人，《Docker 源码分析》作者

黄勇的这本书从微服务实操的角度，通过在微服务架构体系的不同关注点，选择多样而务实的技术栈，为大家全方位地阐述了微服务架构体系的各种最佳实践，对微服务感兴趣的同学不容错过。

—— 王福强，征数科技 CTO，《Spring Boot 揭秘》和《Spring 揭秘》作者

微服务架构，虽然诞生时间不长，却已成为软件架构领域讨论的热点。微服务的概念看似简单，但涉及诸多方法论和实践积累，这就是为什么有人说它非常好，但就是“玩不起”。随着微服务生态系统的日趋完善，微服务架构的讨论也从 API 接口、服务间通信、接口测试、基础设施自动化等，逐渐扩展到了 API 网关、微服务的注册与发现、Docker 封装与部署、持续交

付以及运维体系的优化等多方面。本书结合作者过去多年的实战经验，深入浅出地梳理了微服务构建过程中遇到的诸多挑战，并给出了切实可行的解决方案（如何使用 Spring Boot 构建服务、使用 ZooKeeper 注册服务，如何结合 Docker 封装服务和发布服务等），是一本能帮助读者立刻动手、落地微服务的好书。同时，作者从开发和运维两个角度入手，详细地剖析了微服务实施过程中，如何有效解决“最后一公里”的部署以及运维难题。纵览全书，条理清楚，图文并茂，理论结合实际，是一本非常用心，又注重实操的好书，对企业的微服务架构实施，具有很大的参考意义，相信企业的架构师、软件开发人员、运维人员读完这本书一定会受益匪浅。

—— 王磊，DevOps 教练，《微服务架构与实践》作者

微服务是近几年的一大热点，其模块化、跨语言和自治隔离等思想，有望大幅降低研发和运维成本。微服务架构，无论对传统企业，还是互联网公司，都会有很大影响。黄勇老师结合了 Spring Boot、Jenkins 和 Docker 等热点技术，对微服务的整个生命周期做了全面介绍，通俗易懂、深入浅出，致力于打造微服务领域最佳实践，不失为一本好书。

—— 吴其敏，携程框架研发部高级总监，开源分布式实时监控系统 CAT 作者

当今，微服务已经不是概念，而是势不可挡的潮流，它在大型互联网电商类企业已有丰富的实践，效果很好。但对于其他有志于向微服务架构转型的技术爱好者，微服务如何落地还存在很多不清楚的地方，本文从细节入手，结合具体实例，娓娓道来，为大家提供一个很好的微服务实践参考，带领大家走进微服务之门。

—— 王庆友，独立架构顾问，《架构的本质》作者

软件开发从来没有银弹，微服务也不是。我认为微服务本质上是要解决一个可伸缩性的问题，以应对访问的增加、业务复杂度的增加和开发团队人员的增加。黄勇在这本书中详细解释了实践微服务必须要面对的架构模式，包括服务注册与发现、API 网关以及简单部署系统的搭建，并辅以样例代码，对于正面临可伸缩性问题的开发人员有很大的参考价值。

—— 许晓斌，阿里巴巴高级技术专家，《Maven 实战》作者

近年来，软件开发领域的新思想、新方法、新工具、新实践层出不穷，简直有令人应接不暇、目眩神迷的感觉。要想走出这团迷雾，微服务是纲，容器化、自动化运维、自动化部署、服务监控与治理等，都是目。通过阅读本书，纲举目张，则一切将尽在掌握！

—— 庄表伟，华为内源平台架构师，《开源思索集》作者

随着移动互联网的崛起，Web 网关越来越重要，本书从 Web 网关的视角带领大家学习微服务架构。通过本书可以学习到如何使用 Spring Boot 与 Docker 等技术构建 Web 型微服务架构，值得 Web 开发人员学习。

—— 张开涛，“开涛的博客”博主，《亿级流量网站架构核心技术》作者

微服务是最近几年在架构方面比较热的一个话题，本书从概念到具体的落地，比较系统地介绍了微服务从构建到部署等环节的知识和具体方案，是想了解和学习微服务相关技能的一本好书。

—— 曾宪杰，美丽联合集团副总裁，《大型网站系统与 Java 中间件实践》作者

读者评价

面对近几年火热的微服务架构，但很多人都是「只可远观，不可近玩」的态度，大家对吨级的概念与方法论往往都是望而生畏。他们会觉得微服务架构是复杂的、高端的、赶潮流的、没有必要的，中小团队没有必要也没有实力去落地。本书将许多晦涩与难以理解的概念和方法论用通俗易懂的文字来描述，非常接地气，实在是不可多得的一本微服务架构经典入门书籍。阅读完本书，会有一种「微服务架构也没有想象中的恐怖嘛」的感觉。勇哥手把手地带着读者把轻量级微服务架构落地，完全是从 0 到 1，以及教给读者如何应对里面的坑。在阅读的时候，有一种勇哥在带着我一起在微服务的世界中探险的感觉，作为我的引路人，非常有安全感。本书的内容都是奔着落地去的，不会有飘在天上的各种方法论，诚意满满的干货，绝对物超所值。勇哥的《架构探险》系列书籍是 Java 从业人员在职场中进阶的宝典。所以，你值得拥有。

—— 偏头痛杨，Java 技术经理

一份轻量级微服务架构最佳实践的讲义。全书总分形式，第一章先构建了轻量级微服务架构图，然后逐个章节图文结合，并带有清单式的讲解，行文简洁易读，深入浅出。从目录可以看出，重点介绍了微服务基础设施方面的知识，是不可多得的落地实战总结，我会推荐给任何在微服务架构道路上的技术人。

—— 泥瓦匠 BYSocket，特赞开发工程师

微服务（MSA）是目前企业级应用主流架构和落地方向，黄岛主（作者雅称）与其团队站在微服务思想、架构的高度，澄清了微服务理念 and 原则，通过微服务实践形成行业领先技术栈经验，值得大家品读。黄岛主的《架构探险：轻量级微服务架构》上下册为传统 IT 企业从事微服务实践提供了一套比较完整的微服务方案和指导原则，是传统架构师向微服务架构师转型床头书，是程序员的必读之物。

—— 罗中华，资深架构师

一直有拜读黄勇老师的博客和书籍，非常佩服黄老师能将晦涩难懂的理论用通俗易懂的语言解释清楚，让我们有豁然开朗的感觉。此次黄老师的新书，用实际的案例详解了微服务架构中基础且重要的日志平台、监控中心、配置中心等。读完样章后，希望能立即阅读实体书，干货满满，很多知识点都是现在正在困扰我们的。希望赶快阅读，尽早解决我们的困惑！

—— 彭清正，宝付支付开发工程师

勇哥的书极富启发意义，曾经对我认识、理解架构起到很大的作用。根据本书的上册，很容易推测出下册将对微服务的原理、实践、运维的认知进行一场洗礼。

—— 李阳，Java 与 Golang 开发者

近三年，微服务架构风靡全球，很多互联网企业都在做微服务改造，黄勇老师的《架构探险：轻量级微服务架构（下册）》这书记载了特赞生产环境中微服务实施的具体细节，详细地讲解了微服务实施过程中的日志、通信、消息、分布式事务、配置中心、监控等核心内容，是业界难得的一本微服务架构精品书籍，极力推荐大家阅读。相信每一位读者读完本书必定对微服务架构有着更深入的理解，并且能够快速掌握微服务架构开发。

—— 刘国柱，技术经理

前言

2017 年，微服务三岁了。我们一直期盼它能快速长大，希望微服务技术社区能推出更多框架与工具，可以帮助我们更好地落地微服务，并从中获得微服务给我们带来的甜头，但实际情况却让我们感到有些失望。我们至今还在寻找适合自身技术需求的微服务架构，甚至仍然觉得微服务离自己有些距离。

因此我们更多的是在观望，希望看到有成功实践微服务的企业能够将自己的技术分享出来，以供更多的企业来使用，但实际情况仍然让我们感到失望。我们不得不选择 Spring Cloud 这样的“全家桶”式的微服务框架来实现微服务架构，此时我们就需要基于 Spring Boot 来开发微服务，Spring Cloud 提供的大量基础设施虽然可与 Spring Boot 进行无缝整合，但这样的架构给微服务的技术选型带来了一定的局限性。此外，Spring Cloud 包含了大量的 Spring 官方所提供的开源项目，目前不同的版本在兼容性方面也存在一些不稳定现象。

我们认为，微服务是一个灵活的技术架构，它不能绑定在特定的技术平台上，微服务不应该存在任何的局限性，同时还要确保有较强的兼容性。比如，虽然我们也使用 Spring Boot 开发微服务，但也允许使用其他更适合的开发框架或编程语言来实现微服务。再比如，我们目前通过 ZooKeeper 来实现服务注册，但也能轻松地切换为其他技术选型，对于整个应用程序而言，这些都是无感知的。微服务所提倡的理念就是，用最合适的技术以最高效的方式来解决实际应用中的问题。

经过两年多的实践过程，我们找到了一款能让微服务架构快速且稳定落地的解决方案，并将此方案的核心内容汇集成本书，希望该方案能给微服务世界带一点新的能量。

本书将重点关注微服务基础设施方面，其中大部分内容涉及微服务运维相关技术。全书以实践的角度进行编写，读者首先将学习轻量级微服务架构的全景视图，随后的各个章节将围绕微服务的日志、监控、通信、解耦、测试、配置六大方面进行展开。读者可亲自动手，从零开始搭建轻量级微服务架构，充分享受架构探险的乐趣。

本书是如何组织的?

第 1 章：轻量级的微服务。

本章将从宏观上描述轻量级微服务架构。首先我们将从架构与架构师开始讲起，简单回顾架构演进的过程与微服务的发展趋势。随后我们将探讨在搭建微服务架构之前需要准备的工作，认识微服务架构的“冰山模型”，介绍切分微服务边界的方法和技巧。最后我们将从部署与运行两个角度来观察微服务架构，并以一幅架构全景图来结束本章。

第 2 章：微服务日志。

本章将关注点放在微服务日志上。首先我们将从 Spring Boot 日志框架入手，使应用日志可以输出到 Docker 容器外部，以便我们可随时查看日志文件。随后我们将学习 Docker 日志驱动，使日志信息输出到 Linux 的 Syslog 中。最后我们将 Syslog 与 ELK 技术栈整合，搭建一款微服务的日志中心。

第 3 章：微服务监控。

本章将视角放在微服务监控方面。首先我们将学习 Spring Boot 应用程序自带的监控特性，接着将介绍 Spring Boot Admin 开源监控系统的使用方法。随后我们将集成 InfluxDB、cAdvisor、Grafana 等开源工具，搭建一款微服务的监控中心。最后我们将学习 Zipkin 工具的使用方法，将其用于微服务的追踪中心。

第 4 章：微服务通信。

本章将围绕微服务之间的通信来展开。首先我们将在 Spring Boot 应用程序中实现基于 HTTP 的同步调用，同时我们也会对比 Spring RestTemplate、OkHttp、Retrofit 等工具的使用方法。随后我们将使用 gRPC 框架实现基于 RPC 的同步调用，并将 gRPC 与 Spring Boot 进行整合。最后我们将亲自动手，搭建一款轻量级分布式 RPC 框架。

第 5 章：微服务解耦。

本章将使用消息队列的异步方式来解耦微服务调用问题。首先我们将对比 ActiveMQ 与 RabbitMQ，它们是两款经典的开源消息队列。随后我们将使用 RabbitMQ 来实现请求应答模式，并通过 RabbitMQ 来实现 RPC 同步调用。最后我们将使用 Event-Sourcing 与 MQ 相结合，巧妙地解决分布式事务问题。

第 6 章：微服务测试。

本章将聚焦在微服务测试方面。首先我们将以 Spring Boot 应用程序为例，分别针对 Service 层与 REST API 进行单元测试。随后我们将使用 Postman 来充当 REST API 的测试工具，并结合

Jenkins 与 Newman 搭建一款 REST API 的自动化测试框架。最后我们将分别使用 Swagger 与 apiDoc 工具来自动生成 REST API 文档，并比较这两款工具的优缺点。

第 7 章：微服务配置。

本章将解决微服务的配置参数问题。首先我们将通过一些实例，快速学习 Ansible 自动化运维工具的使用方法。随后将 Ansible 作为微服务的配置中心，并将 Jenkins 与 Ansible 相结合，可用于优化我们现有的微服务部署框架。最后我们将使用 Registrator 所提供的自注册特性，实现微服务的平滑升级目标。

如何获取本书源码？

可通过以下链接下载本书源码。

<http://git.oschina.net/huangyong/msa-book-2>

如何参与线上互动？

欢迎加入“轻量级微服务架构”QQ 群，申请加入时请注明“架构探险”。

群号：528265294

此外，请关注“架构探险图书”微信公众号，可免费获取关于本书后续的更多内容。



致谢

首先需要感谢的是我们的技术团队，如果没有你们所提供的实践经验，这本书也不可能问世。我很庆幸自己能加入特赞技术团队，也很感激你们对我一如既往的支持与鼓励。在微服务实践这条路上，你们才是专家，我只是把你们的宝贵经验整理出来，希望能让更多的人从中受益，我想这也是我们特赞技术团队的共同理想。

自从去年 9 月上册出版以后，很多读者都在问“下册什么时候出版？”，我原本以为是今年上半年就能完成的事情，没想到计划不如变化，公司有更加重要的工作需要我去完成，因此写书的计划也被频繁打断。直到一年后的今天，下册才能与你们见面。在此，我先对你们说声抱

歉，也感谢你们一直对我的关注与等待，希望下册能给你们带来更多的帮助。

最近这一年中，我在很多公开场合下分享过关于微服务的话题，在此向曾经帮助我的技术专家们致谢，和你们聊技术是一件兴奋的事情，让我更加深刻地认识到微服务的本质，也让我对我们技术团队现在搭建的微服务架构更有信心了。同时也感谢你们对这本书所做的推荐，希望我的全力以赴能够配得上你们的称赞。

如果说写完一本书全是作者的功劳，那就大错特错了。没有好的出版社，没有优秀的编辑，我想再好的书也会让大家失望。感谢本书编辑陈晓猛先生对本书的辛勤付出，我在晓猛身上学会了谨慎与专注，这是我和晓猛第三次合作，每次合作都能让我感到，写书其实是一件快乐的事情，我很享受这个过程。

最后我想把感谢的话留给我的妻子和女儿，感谢你们一路陪伴着我，我们共同见证着对方的成长。这本书能够顺利完成，绝对离不开你们努力，因为你们给了我一个幸福而美满的家庭，让我能够将心思沉醉在写作之中。虽然最后我才感谢你们，但你们在我心中永远是最重要的人，永远无法取代。

黄勇

2017年7月18日于上海

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **下载资源：**本书如提供示例代码及资源文件，均可在 [下载资源](#) 处下载。
- **提交勘误：**您对书中内容的修改意见可在 [提交勘误](#) 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动：**在页面下方 [读者评论](#) 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/32447>



目录

第 1 章 轻量级的微服务	1
1.1 微服务将变得轻量级	2
1.1.1 架构与架构师	2
1.1.2 架构演进过程	4
1.1.3 微服务架构发展趋势	6
1.2 微服务架构前期准备	7
1.2.1 认识微服务架构冰山模型	7
1.2.2 冰山下的微服务基础设施	8
1.2.3 根据业务切分微服务边界	9
1.3 轻量级微服务架构图	10
1.3.1 轻量级微服务部署架构	10
1.3.2 轻量级微服务运行架构	11
1.3.3 轻量级微服务全局架构	12
1.4 本章小结	13
第 2 章 微服务日志	14
2.1 使用 Spring Boot 日志框架	15
2.1.1 使用 Spring Boot Logging 插件	15
2.1.2 集成 Log4J 日志框架	17
2.1.3 将日志输出到 Docker 容器外	19
2.2 使用 Docker 容器日志	19
2.2.1 Docker 日志驱动	20
2.2.2 Linux 日志系统：Syslog	23

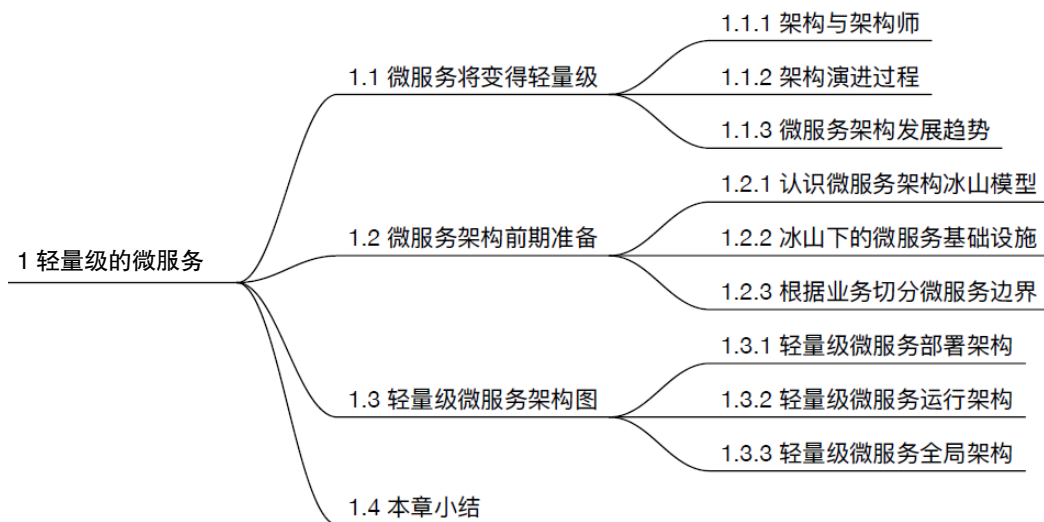
2.2.3	Docker 日志架构	26
2.3	搭建应用日志中心	28
2.3.1	开源日志中心：ELK.....	28
2.3.2	日志存储系统：Elasticsearch	29
2.3.3	日志收集系统：Logstash.....	38
2.3.4	日志查询系统：Kibana.....	44
2.3.5	搭建 ELK 日志中心	46
2.4	本章小结	50
第 3 章	微服务监控	51
3.1	使用 Spring Boot 监控系统	52
3.1.1	Spring Boot 自带的监控功能	52
3.1.2	Spring Boot Admin 开源监控系统	53
3.2	搭建系统监控中心	63
3.2.1	时序数据收集系统：cAdvisor.....	63
3.2.2	时序数据存储系统：InfluxDB	66
3.2.3	时序数据分析系统：Grafana.....	71
3.2.4	集成 InfluxDB + cAdvisor + Grafana	73
3.3	搭建调用追踪中心	78
3.3.1	开源调用追踪中心：Zipkin.....	79
3.3.2	追踪微服务调用链	82
3.3.3	追踪数据库调用链	98
3.4	本章小结	104
第 4 章	微服务通信	105
4.1	使用 HTTP 实现同步调用.....	106
4.1.1	使用 Spring Boot 开发服务端.....	106
4.1.2	使用 Spring RestTemplate 开发客户端.....	108
4.1.3	使用 OkHttp 开发客户端	114
4.1.4	使用 Retrofit 开发客户端	118
4.2	使用 RPC 实现同步调用	127
4.2.1	RPC 通信原理.....	127

4.2.2	初步体验 gRPC.....	128
4.2.3	Spring Boot 集成 gRPC	137
4.3	搭建分布式 RPC 框架.....	144
4.3.1	架构设计	144
4.3.2	搭建模块代码框架	149
4.3.3	开发 RPC 服务端.....	157
4.3.4	开发 RPC 客户端.....	177
4.4	本章小结	186
第 5 章	微服务解耦	187
5.1	使用 MQ 实现异步调用	188
5.1.1	使用 ActiveMQ 实现 JMS 异步调用	188
5.1.2	使用 RabbitMQ 实现 AMQP 异步调用	198
5.2	使用请求应答模式实现 RPC 调用	211
5.2.1	请求应答模式简介	211
5.2.2	使用 RabbitMQ 实现 RPC 调用	214
5.2.3	封装 RabbitMQ 的 RPC 代码框架.....	219
5.3	解决分布式事务问题	225
5.3.1	什么是 Event-Sourcing	225
5.3.2	使用 Event-Sourcing 与 MQ 实现分布式事务控制	227
5.4	本章小结	242
第 6 章	微服务测试	243
6.1	使用 Spring Boot 单元测试	244
6.1.1	搭建待测应用程序框架	244
6.1.2	测试 Service 层	250
6.1.3	测试 REST API	257
6.2	搭建 REST API 自动化测试框架.....	263
6.2.1	使用 Postman 手工测试 REST API.....	264
6.2.2	使用 Newman 批量测试 REST API	272
6.2.3	搭建 REST API 自动化测试框架.....	274
6.3	自动生成 REST API 文档.....	276

6.3.1	使用 Swagger 生成 REST API 文档.....	276
6.3.2	REST API 文档的另一选择: apiDoc.....	285
6.4	本章小结.....	289
第 7 章	微服务配置.....	290
7.1	Ansible 入门与实战.....	291
7.1.1	Ansible 是什么.....	291
7.1.2	准备 Ansible 实战环境.....	292
7.1.3	Ansible 实战.....	293
7.2	搭建服务配置中心.....	306
7.2.1	如何管理微服务中的配置.....	306
7.2.2	设计 Ansible 配置中心.....	308
7.2.3	动手实现自动化部署框架.....	309
7.3	自注册服务配置.....	316
7.3.1	目前服务注册存在的问题.....	316
7.3.2	使用 Registrator 实现服务自注册.....	317
7.3.3	微服务平滑升级解决方案.....	320
7.4	本章小结.....	322

1 chapter

第 1 章 轻量级的微服务



1.1 微服务将变得轻量级

架构需要由人去设计，这些人被称为架构师。或许很多人并未授予架构师的头衔，但自己却从事着架构的工作。我们认为，架构这项工作永远都需要由人去完成，可能短期内都无法由机器来取代。如果我们不理解什么是架构，或者对架构师的职责感到疑惑，那么很难让架构这项工作有效地落地。我们将在本节重新认识架构，并重新定义架构师的职责。此外，架构演进是一个曲折的过程，但我们却不难看出架构的发展规律，甚至还能推测出架构将来的发展趋势。我们相信，微服务一定不是架构的终点，它或许只是架构从重量级转型为轻量级的桥梁，我们正是设计并建造这座桥梁的工程师。

现在我们从架构与架构师的角度开始出发，开启轻量级微服务的架构探险之旅。

1.1.1 架构与架构师

可能绝大多数的程序员都想成为一名优秀的架构师，每天都能从事技术架构的相关工作，编点框架代码，画点架构图，写点 PPT，帅气地站在讲台上给程序员们进行技术培训。大家普遍认为，架构师的代码比别人写得少，但是工资却比别人拿得多，架构师是技术团队中技术最牛的人，别人搞不定的技术问题，在架构师眼中都是小菜一碟。

这样的人真的是架构师吗？

我们认为，他们不是架构师，而是技术专家。其实架构师与技术专家有着本质的区别，从他们所关注的技术方向来看，架构师更偏向技术广度，而技术专家更偏向技术深度，如图 1-1 所示。

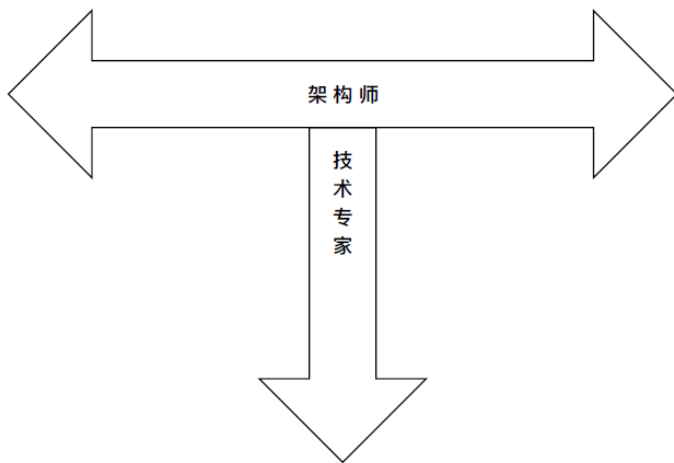


图 1-1 架构师与技术专家的区别

换句话说，架构师需要有较强的综合能力，他们需要接触的技术领域较广，但他们所掌握的技术专业能力却没有技术专家那么深。如果我们想成为一名架构师，那么就不应该把所有的精力都投入在某个技术领域上，而是要学会分散自己所关注的层面，做到在众多技术领域上都要有一定的深度。

架构师除了需要具备在技术上所需的“硬技能”，还需要不断完善自己的“软技能”，比如沟通、组织、学习等技能。有时候软技能可能比硬技能更加重要，甚至软技能还会影响自己的职业发展。如果没有较好的软技能，架构师将无法将自己所设计的架构顺利地移交到程序员们手中，并指导他们将其真正落地。架构师正是通过他们所具备的综合能力来带领技术团队，解决不断出现的技术挑战。

架构师的职责是什么？

我们的回答是：制定规范 + 指导落地。

架构师根据业务需求所制定的合理且可落地的技术规范，我们将这样的规范称为架构。

将架构工作做好犹如我们用两条腿走路一样，左腿迈出去表示“制定规范”，右腿跟上来表示“指导落地”，如图 1-2 所示。

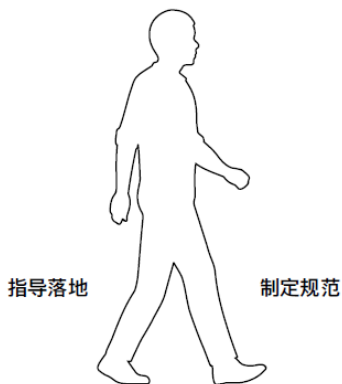


图 1-2 架构师的职责

如果左腿迈出去，右腿没跟上来，那不是架构师，可能是需要拄拐的人。然而，我们身边却有一些这样的不合格的架构师，他们只懂得制定规范，却忽略了指导落地。如果架构无法落地，那么就无法称为架构了。

此外，还有一些架构师认为，架构只是技术层面的问题，自己设计的架构应该用到市面上最为流行的新技术，比如别人公司在用微服务，那么自己公司也要用起来。如果将架构工作脱离于业务需求，我们认为这不是做架构，而是玩技术。脱离业务来设计架构是对架构的不尊重。微服务是一种应用系统架构，需要架构师围绕业务进行设计。

但是，我们绝不要为了微服务而去微服务。

从事微服务架构工作的架构师，相比传统架构的架构师而言，所要求的技能更加全面。他们不仅仅是系统架构师，也是业务分析师，他们的责任重大且挑战艰巨。

从大的方向来看，微服务架构师需要具备以下基本职责。

- (1) 分析业务需求并切分微服务边界。
- (2) 定义架构规范与文档标准。
- (3) 确保微服务架构顺利落地。
- (4) 改善微服务架构并提高开发效率。

职责与挑战往往是无法分离的，微服务架构师必须面对并克服这些挑战。

- (1) 架构需要适应不断变化的业务需求。
- (2) 架构具备稳定性、扩展性、安全性、容错性等。
- (3) 使技术团队深刻理解微服务思想。
- (4) 展现微服务架构的价值。

我们认为，传统架构师转型为微服务架构首先需要做到的是深刻理解业务，而不是表面上了解需求。业务和需求其实是两码事，业务的背后反映了客户的刚需，而需求往往是产品经理根据业务刚需所指定的解决方案。作为微服务架构师，我们需要透过需求的表象去理解业务的本质。其次需要做到的是不断优化架构，让架构变得更加简单，更加轻量级。我们要将昨天最好的表现，当成今天最低的要求，只有在技术上不断要求自己，才能让架构变得更好。

1.1.2 架构演进过程

话说天下大势，分久必合，合久必分，对于架构演进过程而言，也符合这个规律。

最早的应用程序实际上是没有任何架构的，因为那时业务比较简单，没有架构也许是合理的，如图 1-3 所示。

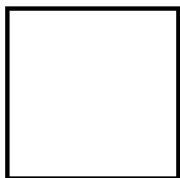


图 1-3 没有架构

但随着业务不断复杂起来，我们意识到架构可以做到水平分层，比如表现层、逻辑层、数

据层等，我们可在不同的层上实现每一层所关注的内容，我们称其为“关注点分离”。但此时的架构更像是一块“铁板”，每一层的无法进行分离，因此我们也将这样的架构称为“单块架构”，如图 1-4 所示。

从此架构发生了更为复杂的变化，层次结构越来越深，而且不再局限于水平方向上的分层，实际上越来越多的应用程序围绕着不同的业务需求来实现，此时出现了“垂直分层架构”，每个垂直应用中实际上都是一个独立的子系统，它们共同组成了整个应用系统。然而，这些子系统一般可以部署在不同的服务器上，这些服务器可以分布在不同的地域中，我们也称其为“分布式架构”，如图 1-5 所示。

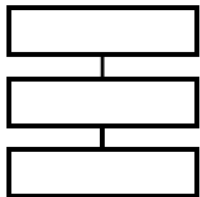


图 1-4 水平分层架构（单块架构）

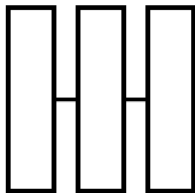


图 1-5 垂直分层架构（分布式架构）

业务并没有停止发展的脚步，架构的演变也是如此。分布式应用之间的调用越来越多，整个系统的复杂度急剧上升，人们希望找到一种途径来降低分布式应用之间的耦合。此时出现了面向服务架构（SOA），人们希望 SOA 能成为解决分布式应用系统复杂性的“银弹”，然而事实却事与愿违。应用的复杂性不仅没有得到解决，反而还让架构变得更加复杂，同时也形成了大量的 SOA 商业产品，这些现象让人们更加恐惧 SOA，将它视为复杂和昂贵的代名词，如图 1-6 所示。

随着互联网行业不断发展，用户对产品的体验要求越来越高，前端的价值逐渐被凸显出来，此时出现了“前后端分离”的趋势，前端工程师专心在界面展现与数据渲染上，后端工程师关注在业务逻辑与数据结构上，前后端只需通过 REST API 进行交互，工作分工更加明确，开发效率更加高效，如图 1-7 所示。

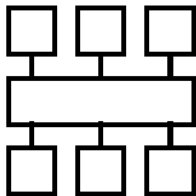


图 1-6 面向服务架构（SOA）

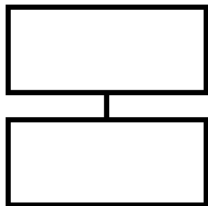


图 1-7 前后端分离架构

似乎很长时间都未出现任何技术能与当年的 SOA 相提并论，除了微服务架构。微服务的概念在 2014 年首次被提出以来，近几年一直是应用架构领域的核心话题。但人们往往还是容易想

到当年的 SOA，认为微服务与 SOA 有着相同的目的，只是实现细节不太一样罢了，如图 1-8 所示。

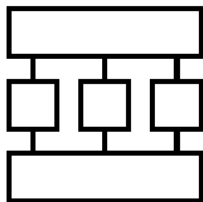


图 1-8 微服务架构

微服务与 SOA 到底有何区别？

我们认为，微服务是 SOA 的一种落地方案。

SOA 是一种面向服务的架构思想，微服务也同样推崇这种思想。微服务是将一个大型的单块架构，拆分为多个细粒度服务的架构。微服务更加考验我们的是，深入理解业务并合理地对服务边界进行切分。微服务的概念相比 SOA 更容易落地的原因不是概念上的创新，而是技术上的突破，尤其是容器与自动化运维技术的普及与应用。

我们始终相信，微服务并不是架构发展的终点，也许是新架构时代的起点。

1.1.3 微服务架构发展趋势

微服务架构所涉及的范围相当广泛，我们不妨从多个角度来推测微服务架构的发展趋势。

从微服务开发角度来看，我们认为微服务的开发框架将变得更加多样化。

开发人员可使用更加适合的开发框架来完成微服务业务实现，而不再拘泥于某一种编程语言，只需确保对外提供统一的 API 接口方式即可。甚至可将查询与修改操作相分离，查询操作可以用一种更加轻量级的编程语言来实现，而修改操作会涉及事务，一般需要借助开发框架的事务特性来保证。

我们坚信，微服务必将坚持走轻量级技术路线。

究竟什么是轻量级？

我们认为，轻量级必须包含三个特征：易用、快速、稳定。

我们希望微服务架构中所涉及的技术都能够快速上手，运行时不占用过多的系统资源且性能突出，而且能够长期稳定地运行。

从微服务部署角度来看，我们认为微服务的部署过程将变得更加自动化。

部署微服务不再通过手工的方式去完成，因为这样既低效又容易出错，我们更加倾向于使

用软件工具将其自动完成。要实现自动化部署这个目标，我们往往无法一步到位，最合理的方式是“先让它跑起来，再让它跑得快”。也就是说，早期的自动化部署方案也许不够完备，或多或少会存在一些人工参与的情况，其实这些都再正常不过了，但我们需要不断优化，努力通过自动化技术来取代重复性的人工操作。

最后想说的是，自动化虽好，但不要为了自动化而去自动化，或许有些环节通过手工处理才是最有效的方式。

1.2 微服务架构前期准备

搭建微服务架构绝不是一件轻松的事情，我们不仅要對微服务概念有着深刻的理解，还要研究大量的技术工具，并掌握它们的优缺点，最终需要结合技术团队对技术的了解程度，选择最为合适的技术选型。这些纯技术的技术工具只是微服务的基础设施，我们还需要在此基础设施上围绕真实的业务场景，对微服务边界进行合理地切分。我们将在本节介绍微服务的冰山模型，大家将从这座冰山中看到微服务架构的全貌，随后我们将深入冰山之下的世界，去探索微服务基础设施的八大中心，最后我们还会介绍一些关于切分微服务边界的原则和技巧。

我们现在就从微服务冰山模型开始吧，这座冰山似乎比我们想象中的要大很多。

1.2.1 认识微服务架构冰山模型

有些人认为，使用了 Spring Boot 开发框架就是拥有了微服务，其实这样的认识是不正确的。Spring Boot 只是一款微服务的开发框架，而且仅能用于 Java 应用程序中，毫无疑问，它只是微服务的冰山一角。此外，我们建议大家结合自身的业务场景，选择更为合适的编程语言以及开发框架来实现微服务，而不要拘泥于一种编程语言。

还有一些人认为，用上了 Docker 就进入了微服务时代，其实这样的认识也是不正确的。Docker 只是一种封装微服务应用程序的容器化技术，它改变了应用程序的交付方式，也加速了微服务架构的落地速度。可以肯定地说，如果没有 Docker 容器技术，或许今天我们无法听到微服务的概念。

如果将微服务架构中所涉及的技术栈比喻为一座冰山的话，那么冰山之上最显而易见的部分就是微服务的开发框架与容器技术了，Spring Boot 与 Docker 都属于冰山之上的技术。

冰山之下到底有哪些技术呢？

我们认为冰山之下的技术是整个微服务架构的基石，它们构成了整个微服务架构的基础设施。比如我们在上册中学习的 ZooKeeper 服务注册表、Node.js 服务网关、Jenkins 持续部署系统等，它们都属于冰山之下的部分。

我们可通过一幅图来描绘微服务架构这座冰山，如图 1-9 所示。

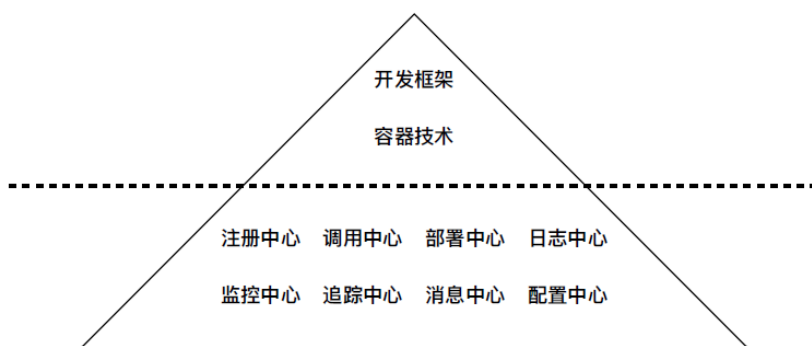


图 1-9 微服务架构冰山模型

由于服务注册表集中管理了微服务相关的服务配置，因此我们也将它称为“注册中心”；由于服务网关是前端应用程序的唯一入口，因此我们也将它称为“调用中心”；同样，我们也将持续部署系统称为“部署中心”。这些中心都汇集在微服务冰山模型之下，除此之外，还有其他功能的中心，这些中心共同构成了微服务的基础设施。

1.2.2 冰山下的微服务基础设施

冰山下的微服务基础设施，实际包括了八大中心。

- (1) 注册中心：用于注册微服务相关配置信息的中心，我们选用 ZooKeeper 实现。
- (2) 调用中心：用于提供给前端调用的统一入口，我们选用 Node.js 实现。
- (3) 部署中心：用于编译并打包微服务源码并将其部署到 Docker 引擎中，我们选用 Jenkins 实现。
- (4) 日志中心：用于收集并管理微服务应用程序中产生的日志，第 2 章中将详细介绍。
- (5) 监控中心：用于监控微服务的实时运行状况，第 3 章中将详细介绍。
- (6) 追踪中心：用于最终微服务的调用轨迹，第 3 章中将详细介绍。
- (7) 消息中心：用于解耦微服务之间的调用关系，第 5 章中将详细介绍。
- (8) 配置中心：用于管理微服务应用程序所需的配置参数，第 7 章中将详细介绍。

也许大家看到以上八大中心后会产生一些疑惑：很多人说微服务是去中心化的，为何我们还要提供这些中心呢？

我们认为，中心分为两类：一类是含有业务意义的中心；另一类是不含业务意义的中心（只是技术层面的中心）。

在微服务架构中我们应该去除的是含有业务意义的中心，而不是去除技术层面上的广义中心。比如，我们所设计的调用中心内部是不可能带有任何业务流程的，它只是一个纯技术层面的框架。对于其他中心也是如此，绝对不含有任何的业务，否则我们就应该将其去除。

1.2.3 根据业务切分微服务边界

凡是学习过微服务的人都知道，我们需要根据业务来切分微服务边界。道理可能大家都懂，但或许仍然不知道应该怎么做。例如，切分微服务边界有哪些关键性步骤，以及包含哪些重要性原则？

经过大量的微服务实践，我们总结了以下五个步骤，可帮助大家有效地切分微服务边界。

第一步：梳理业务流程。

在切分微服务之前，我们要做的第一件事情就是梳理业务流程。不妨找业务专家咨询，通过与他们沟通从而了解真实的业务流程，并将其绘制成流程图。对于过于复杂的业务流程，我们也可单独绘制流程图，并增加相关的流程说明。当然也能提供相应的状态图，用于说明业务流程中所涉及状态的变化过程。

花再多时间去分析业务流程都不过分，现在所花的每一分钟都是相当值得的。

第二步：抽取公共服务。

在业务流程中与业务不太相关的部分，我们可考虑将其剥离出来，并形成公共服务。例如，邮件发送、文件上传、其他第三方接口等。每种公共服务都对应一个微服务，每个微服务都有相关 API，每个 API 都有自己的输入与输出。这些 API 一定要形成文档，以便其他服务调用。

一般情况下，抽取的公共服务都不太会变化，我们一定要想办法将不变的东西从可变的世界中抽取出来。

第三步：定义业务服务。

当公共服务抽取完毕后，业务流程中剩下的部分就是业务服务了。建议刚开始实施微服务时，不要将业务服务的边界切得太细，可以考虑先“大切几块”，但需要确保每个服务之间尽量不要有依赖关系。换句话说，每个服务都是独立的，虽然此时服务的块头可能比较大。

我们先确保这些大块头服务可以运行在微服务基础设施上，再不断将它们进行细化，拆解为更小的服务。

第四步：设计数据模型。

深入到每个业务服务中，我们首先要做的是定义它底层所涉及的数据模型，也称为“领域模型”。此时会涉及数据库表结构设计，以及数据模型与关系设计。在数据层面上的设计是至关

重要的，如果该部分设计得不到位，将增加后期实现微服务的成本。

数据模型的设计同样也需要进行文档化，这些文档将指导后端工程师顺利地完微服务实现。

第五步：定义服务接口。

底层的数据模型设计完毕后，我们将视角转换到顶层的服务接口上。服务接口实际上就是一组 API，这些 API 需做到职责单一，而且需要通过名称就能识别出它的业务含义。建议确保每个 API 的命名是全局唯一的，也建议每个 API 都有各自的版本号，版本号可以用自增长的方式来体现。

服务接口也需要进行文档化，这些文档一般由后端工程师编写，并提供给前端与测试工程师阅读。

1.3 轻量级微服务架构图

作为一名微服务架构师，我们不仅需要深入理解业务，并能准确地划分服务边界，而且还需要深入理解微服务，并能合理地选择技术选型。我们认为，微服务架构实际上分为两大部分，其中一部分对应部署阶段，另一部分对应运行阶段。这两部分包含了大量的技术工具，我们需要结合多方面因素来考虑，选择最为合适的技术选型来搭建微服务架构，并确保它能保持轻量级。

本节将着眼于微服务的部署与运行两大阶段，通过图文方式来描述轻量级微服务架构。

1.3.1 轻量级微服务部署架构

当开发人员完成了微服务的细节实现后，首先要做的是确保自己所写代码的可用性，他们往往会借助单元测试工具来保证这一环节不出问题。当他们将源码提交并推送到代码仓库后，此时部署中心将从代码仓库中获取源码，并执行编译与打包操作。

不仅如此，部署中心还需从配置中心获取对应运行环境的配置参数，并生成相应的配置文件，并将这些配置文件与应用程序一同复制到 Docker 镜像中，最后还需将此镜像上传到镜像仓库，以便后续可从镜像仓库中下载指定的镜像，从而运行相应的 Docker 容器。

此外，部署中心还可扫描源码并自动生成 API 文档站点，以便其他技术人员可随时从文档站点中查看最新部署服务所包含的 API 文档。当然，我们还可以对所需完成的微服务仅提供简单的代码实现，这样就能将微服务文档的输出时间尽可能提前，以便其他技术人员能够更早地了解微服务的相关 API 信息，随后再去完成更加细节的代码实现，这是我们推荐的工作方法。

当 Docker 镜像上传到镜像仓库后，部署中心可在不同的运行环境下根据特定的镜像来启动相应的 Docker 容器。为了便于描述，我们将该容器称为“服务容器”，它包含了承载微服务的应用程序及其配置文件。

当服务容器启动后，会自动将其配置信息写入注册中心，与此同时，部署中心也会连接到注册中心，并设置服务的版本号，以便于在后续调用服务时可根据版本号来识别当前可用的服务。

我们可将以上过程绘制为一幅架构图，如图 1-10 所示。

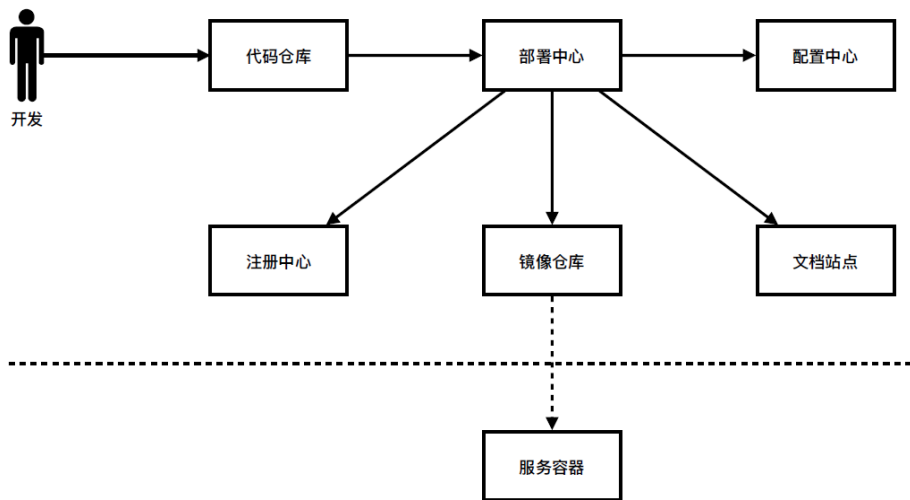


图 1-10 轻量级微服务部署架构

可见，在微服务部署阶段中，部署中心是其中的主角，它支配其他组件，使服务可以成功部署，我们需要确保它的稳定性。

1.3.2 轻量级微服务运行架构

当用户通过浏览器或移动端访问应用系统时，请求首先将进入服务网关，因为它是所有请求调用的中心，我们也将其称为“调用中心”。它虽然是不带任何业务的中心，但我们需要确保它所做的事情足够少，让它不会成为整个应用系统的调用瓶颈。

随后调用中心将连接注册中心，并通过服务名称从注册中心中获取服务所在的 IP 地址与端口号（即服务地址），该过程称为“服务发现”，进而调用中心可根据服务地址以反向代理的方式来调用具体的服务容器，该过程称为“服务调用”。

在服务容器中可能会触发一些事件，这些事件将以消息的方式写入消息中心，以便其他服务可监听消息中心并从中获取相应的消息。该方案可解决服务之间的耦合问题，同时能将同步

调用转为异步调用，提高整个应用系统的吞吐率。

在服务容器运行时会产生大量的日志，我们可将这些日志统一写入日志中心，并能在日志中心所提供的控制台上查询具体的日志信息。此外，日志中心也能帮助我们快速地定位并分析系统出现的异常状况。

为了观察服务容器是否运行正常，我们可借助监控中心所输出的图形化数据来判断。监控中心将不断地收集服务容器中的运行状态，包括 CPU、内存、硬盘、网络，以及应用程序的 JVM 内存使用情况。

由于微服务很难切得干净，服务之间难免会出现少量的调用关系，我们可将每次调用所产生的相关信息写入追踪中心，并通过追踪中心提供的图形化界面来查看服务之间的调用轨迹以及所产生的调用延时，从而可分析出服务调用所产生的性能瓶颈。

我们可将以上过程绘制为一幅架构图，如图 1-11 所示。

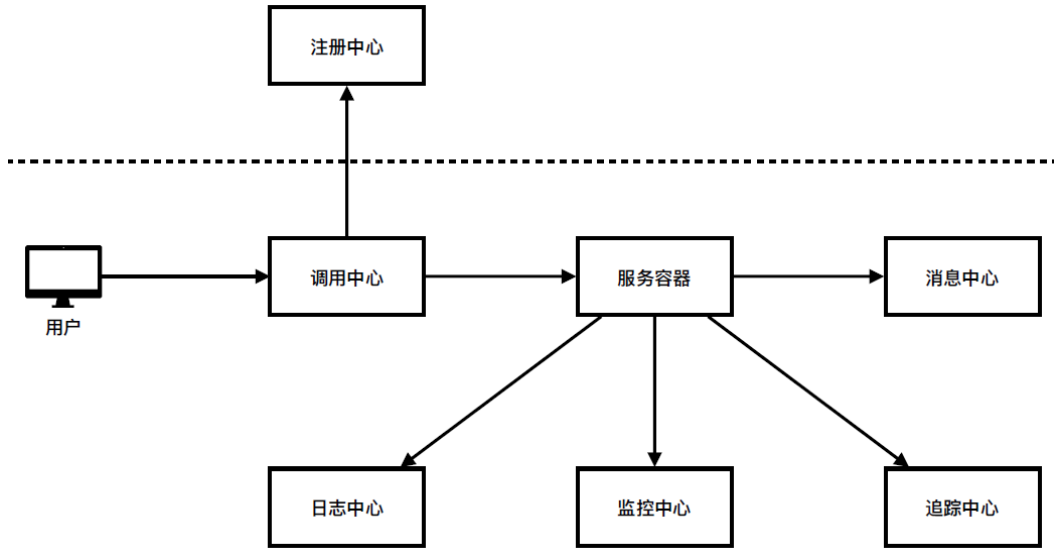


图 1-11 轻量级微服务运行架构

可见，在微服务运行阶段中，调用中心是其中的主角，注册中心作为它的数据来源，服务容器作为它的调用目标，它需要具备良好的高性能与高可用等特性。

1.3.3 轻量级微服务全局架构

我们用一张图将轻量级微服务的部署架构与运行架构进行整合，它就是轻量级微服务架构的全貌，如图 1-12 所示。

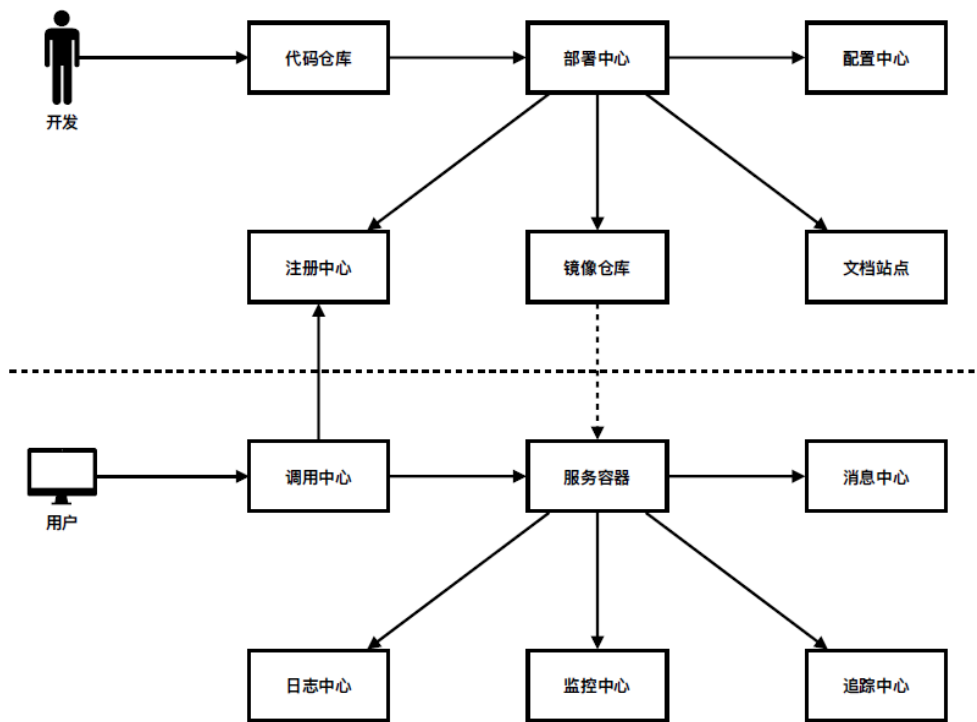


图 1-12 轻量级微服务全局架构

图 1-12 所示的架构图中包括 12 个组件，其中的代码仓库、部署中心、注册中心、镜像仓库、调用中心、服务容器这 6 个组件已在上册中进行描述，剩下的配置中心、文档站点、消息中心、日志中心、监控中心、追踪中心这 6 个组件将在下册后续章节中深入探索。

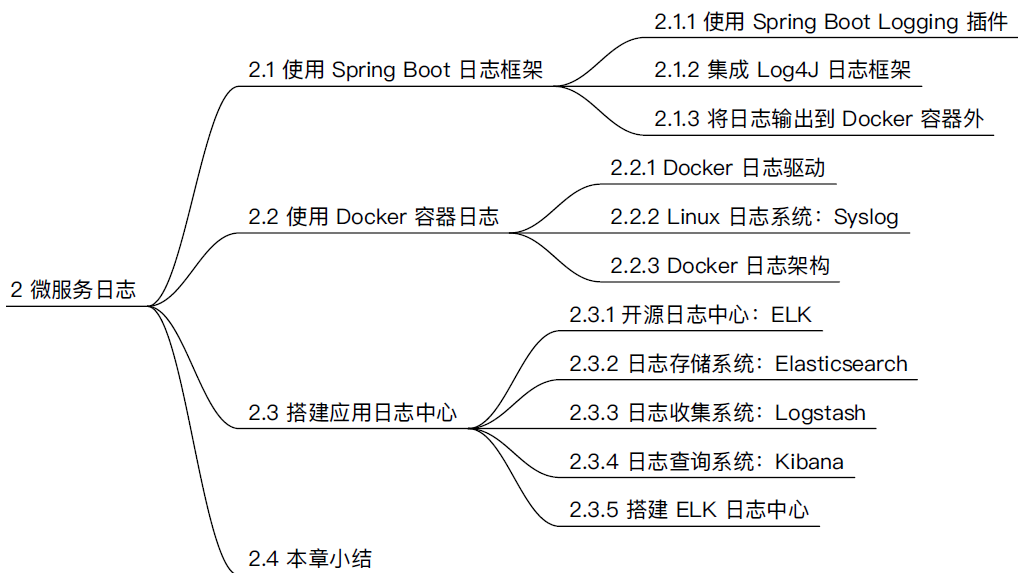
1.4 本章小结

本章从宏观上描述了轻量级微服务架构，为后续探险历程提供了明确的蓝图。首先我们从架构与架构师开始讲起，简单回顾了架构演进的过程与微服务的发展趋势，我们认为，微服务的出现是必然的，而且它将朝着轻量级方向发展。随后我们探讨了在搭建微服务架构之前需要准备的工作，也认识了微服务架构的“冰山模型”，本书将重点集中在冰山之下的微服务基础设施中，此外还介绍了切分微服务边界的方法和技巧，我们认为，合理地切分微服务边界是微服务架构师的职责之一。最后我们从部署与运行两个角度来观察微服务架构，并以一幅架构全景图来结束本章，随后的章节将围绕这张架构图的相关部分进行展开，我们会选择最为合适的技术选型来搭建这款轻量级微服务架构。

下一章我们将重点关注在微服务日志方面，并实现轻量级微服务架构中的“日志中心”。

2chapter

第 2 章 微服务日志



2.1 使用 Spring Boot 日志框架

现在，我们需要在已有的微服务代码中添加日志功能，用于输出需要关注的内容，这是最平常的技术需求了。由于我们的微服务代码是基于 Spring Boot 开发的，那么问题就转换为如何在 Spring Boot 应用程序中输出相应的日志。在传统 Java 应用程序中，我们一般会使用类似 Log4j 这样的日志框架来输出日志，而不是直接在代码中通过 `System.out.println()` 来输出日志。为什么要这么做呢？原因有两点。其一，我们希望日志能输出到文件中，而不是输出到应用程序的控制台中，这样更加容易收集和分析。其二，我们可以通过异步多线程的方式，将日志输出到文件中，这样既不会影响主线程，可以提高应用程序的吞吐率，也是一种节省性能开销的方法。直接将内容打印到控制台中，这种做法比较粗暴，不是业界所推荐的做法。

这样一来，我们最终要解决的问题就非常清楚了，那就是如何在 Spring Boot 中添加日志框架。幸运的是，Spring Boot 自带了一款名为 Spring Boot Logging 的插件（在 Spring Boot 中，称插件为 Starter），它已经为我们提供了日志功能。

2.1.1 使用 Spring Boot Logging 插件

Spring Boot 使用 Apache 开源项目 Commons Logging 作为内部的日志框架，它是一个日志接口，在实际应用中，我们需要为该接口指定相应的日志实现。Spring Boot 默认的日志实现是 Java Util Logging，它是 JDK 自带的日志包，一般场景下很少会用到。此外，Spring Boot 也提供了 Log4J、Logback 这类流行的日志实现，我们只需要添加简单的配置，就能开启对这些日志实现的支持。

为了便于描述，我们将以上提到的“日志实现”统称为“日志框架”。

大家可以通过以下网站，进一步学习这类日志框架。

- Commons Logging 官网：<https://commons.apache.org/proper/commons-logging/>。
- Log4J 官网：<https://logging.apache.org/log4j/2.x/>。
- Logback 官网：<https://logback.qos.ch/>。

在 Java 应用程序中，日志一般分为以下 5 个级别。

- ERROR：错误信息；
- WARN：警告信息；
- INFO：一般信息；
- DEBUG：调试信息；
- TRACE：跟踪信息。

以上日志级别按照严重程度，从高往低排序，一般常用的三种日志级别是 ERROR、INFO、DEBUG。Spring Boot Logging 插件默认输出到 INFO 级别，也就是说，只包含 ERROR、WARN、INFO，不包含 DEBUG、TRACE。如果我们希望日志可以输出到 DEBUG 级别，则需在 Spring Boot 的 application.properties 文件中添加如下配置：

```
logging.level.root=DEBUG
```

重新运行应用程序，我们就可在代码中看到 DEBUG 级别的日志了。

以下是 Spring Boot 的应用程序代码片段，我们使用 SLF4J 类库输出日志，而不要使用具体的日志实现类库，比如 Log4J。

```
package demo.msa;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
...

@RestController
@SpringBootApplication
public class HelloApplication {

    private static Logger logger = LoggerFactory.getLogger(HelloApplication.class);

    public static void main(String[] args) {
        SpringApplication.run(HelloApplication.class, args);
    }

    @GetMapping("/hello")
    public String hello() {
        logger.debug("log..."); // 输出 DEBUG 级别的日志
        return "hello";
    }
}
```

运行以上 Spring Boot 应用程序，会发现控制台中输出了大量 INFO 级别的日志，这些日志是由 Spring Boot 框架输出的。因为我们调整日志输出到 DEBUG 级别，而 INFO 级别在 DEBUG 级别之上，所以 INFO 级别的日志也会输出，但 TRACE 级别的日志不会输出。

当我们打开浏览器，发送 `http://localhost:8080/hello` 请求时，可在控制台中看到我们想要输出的 `DEBUG` 级别日志。

如果我们不想关注 `Spring Boot` 框架的日志，则可将日志级别统一设置为 `ERROR`，此时只会输出 `ERROR` 级别的日志。随后，再将 `Spring Boot` 应用程序指定的包（应用程序所对应的包）设置为 `DEBUG` 级别的日志，此时我们看到的就只是指定包中的日志了。

```
logging.level.root=ERROR
logging.level.demo.msa=DEBUG
```

上面的 `logging.level.root` 表示所有包，`logging.level.demo.msa` 表示应用程序的指定包（`demo.msa` 是包名）。以上配置可以理解为，整个应用程序的日志输出到 `ERROR` 级别，除了 `demo.msa` 包中的日志输出到 `DEBUG` 级别。这是一种“先禁止所有，再允许个别”的配置方法，这种配置方法在很多技术中都应用过。

默认情况下日志框架会将日志输出到控制台中，我们需要在 `application.properties` 文件中添加如下配置，才能将日志输出到文件中：

```
logging.file=${user.home}/logs/hello.log
```

其中，`${user.home}` 表示当前用户目录（该变量由 `Spring Boot` 框架在运行时传入），后面的 `/logs/hello.log` 是相对于该目录的路径。大家可根据实际情况，设置所需的日志文件路径，以上仅为示例。

重新运行应用程序，就能看到日志输出到指定路径下的文件中了。

目前我们虽然可以将日志输出到文件中，但控制台中仍然会输出同样的日志，这不是我们最终想要的效果。我们希望的是日志全部输出到文件中，控制台中不输出任何日志。也就是说，我们需要关闭控制台中的输出。通过以上的尝试，我们不难发现，仅通过修改 `Spring Boot` 的配置，貌似是无法做到的。

下面我们不妨考虑集成经典的 `Log4J` 日志框架，看看能否实现我们的需求。

2.1.2 集成 Log4J 日志框架

`Spring Boot Logging` 默认集成了 `Logback`，我们只需提供 `Logback` 的配置文件就能开启 `Logback` 日志功能，但我们现在想要尝试的是自己熟知的 `Log4J`，而不是比较新潮的 `Logback`。毫不犹豫，现在我们就来开启对 `Log4J` 的支持。通过学习 `Spring Boot` 的官方文档与示例代码，我们了解到，只需在 `pom.xml` 文件中添加如下 `Maven` 配置，就能在 `Spring Boot` 中集成 `Log4J`。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter</artifactId>
  <exclusions>
    <exclusion>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-logging</artifactId>
    </exclusion>
  </exclusions>
</dependency>

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>
```

在第一段 `dependency` 配置中,我们排除掉 `spring-boot-starter-logging` 依赖是因为要去掉默认集成的 Logback 日志功能。在第二段 `dependency` 配置中,我们自行添加了 `spring-boot-starter-log4j2` 依赖,它是 Spring Boot 所提供的 Log4J 插件,此时使用的是 Log4J 的 2.x 版本。

当完成了 Maven 依赖配置以后,我们接下来需要在源码中的 `resources` 目录下添加 `log4j2.xml` 文件,其内容如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <appenders>
    <File name="file" fileName="${sys:user.home}/logs/hello.log">
      <PatternLayout pattern="%d{HH:mm:ss,SSS} %p %c (%L) - %m%n"/>
    </File>
  </appenders>
  <loggers>
    <root level="ERROR">
      <appender-ref ref="file"/>
    </root>
    <logger name="demo.msa" level="DEBUG"/>
  </loggers>
</configuration>
```

`log4j2.xml` 配置文件分为两大部分,即 `appenders` 与 `loggers`。在 `appenders` 中,我们添加了

一个 File 类型的 `appenders`，表示日志以文件的方式进行输出，该文件路径基于根目录 `${sys:user.home}`，即当前用户目录（该变量由 Log4J 框架在运行时传入）。此外，还需指定 `PatternLayout` 为日志输出格式。在 `loggers` 中，我们先后添加了两段配置，第一段的 `root` 表示将所有包中的日志输出到 ERROR 级别，第二段的 `logger` 表示将指定包 `demo.msa` 中的日志输出到 DEBUG 级别。很明显，这段配置与之前在 Spring Boot 中配置的意义相同。

通过以上配置，可将 Log4J 集成到 Spring Boot 应用中。

重新运行应用程序，日志不再输出到控制台中，而是全部输出到指定路径下的文件中了。

大家如果想了解更为详尽的 Spring Boot 日志特性，可参考它的官方技术文档。

Spring Boot 日志：<http://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-logging.html>。

目前，虽然日志已经成功输出到文件中，但是我们的微服务是以 Docker 容器的方式来运行的，此时输出的日志文件仍然和应用程序在一个 Docker 容器中，我们得想办法将日志文件输出到 Docker 容器外。也就是说，需要将数据与程序相分离，以便后续更加方便地获取并分析日志内容。

2.1.3 将日志输出到 Docker 容器外

最容易想到的办法就是，通过 Docker 数据卷的方式，将文件路径挂载到 Docker 容器上，这样日志文件就自然与 Docker 文件分离了，就像下面这样启动 Docker 容器。

```
docker run -v ~/logs:~/logs hello
```

这样一来，我们可随时在宿主机上查看 Docker 容器内部的日志了。但是回过头想想，却不难发现，其实完全不需要将日志输出到文件中，因为即便将日志输出到控制台中，我们也能随时通过 `docker logs` 的方式来获取日志内容，将日志输出到文件似乎有些多余了，还占用了磁盘空间。

为了搞清楚现在做的事情是否有意义，我们不妨先来探索 `docker logs` 背后的秘密吧。

2.2 使用 Docker 容器日志

我们使用 `docker logs` 命令可以随时查看 Docker 容器内部应用程序运行时所产生的日志，这样可以避免首先进入 Docker 容器，再打开应用程序的日志文件的过程。可见，`docker logs` 是一种相当便捷的工具。我们可以这样想象，`docker logs` 会监控容器中操作系统的标准输出设备

(STDOUT)，一旦 STDOUT 有数据产生，就会将这些数据传输到另一个“设备”中，该设备在 Docker 的世界中被称为“日志驱动 (Logging Driver)”。

下面我们先从 Docker 的日志驱动开始探索，一步步揭开 docker logs 的神秘面纱。

2.2.1 Docker 日志驱动

为了便于研究，我们以官方提供的 Nginx 镜像为例，来初步学习 docker logs 的基本技法。

首先我们要做的是，通过以下 Docker 命令启动 Nginx 容器。

```
docker run \
-d \
-p 80:80 \
--name nginx \
nginx
```

我们启动了一个 Nginx 容器，该容器中的 Nginx 进程在后台运行 (-d)，对外暴露了一个 80 端口，同时也映射到容器内部的 80 端口 (-p 80:80)，该容器的名称为 nginx (--name nginx)。

打开浏览器，在地址栏中发送 http://localhost/ 请求 (80 端口可以省略不写)，可以立即看到 Nginx 的首页。

随后我们使用 docker logs 命令，查看 Nginx 容器的日志。

```
docker logs -f nginx
```

我们监控日志尾部(-f)，此后在浏览器中刷新几下，同时观察到 Docker 日志也在同步输出，此时输出的日志类似下面这样。

```
180.166.202.194 - - [09/Mar/2017:03:32:03 +0000] "GET / HTTP/1.1" 304 0 "-"
"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_3) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/56.0.2924.87 Safari/537.36" "-"
```

很明显，这是 Nginx 收集的浏览器信息，此外还包括了客户端 IP、请求时间、请求状态等信息。

只要 Docker 容器内部的应用程序在控制台中有日志输出，就能通过 docker logs 命令来查看相应的日志。Docker 是怎样做到的呢？或者说，所谓的 Docker 日志驱动，到底做了些什么事情呢？我们带着这些问题继续探索。

我们使用 docker info 命令，可以看到 Docker 容器的相关信息，其中有一项叫 Logging Driver

的字段。

```
docker info | grep 'Logging Driver'
```

通过输入以上命令，将得到 Docker 当前所设置的日志驱动类型，它就是 json-file。

顾名思义，json-file 表示 JSON 文件，也就是说，Docker 容器内部的应用程序所输出的日志，将自动写入一个 JSON 文件中。那么这个 JSON 日志文件到底存储在哪里呢？

我们通过学习 Docker 官方文档可知，有一个/var/lib/docker 目录用于存放 Docker 的系统文件。进入该目录，就能看到一个名为 containers 的目录，这个目录中存放的就是当前运行的 Docker 容器，那么这个 JSON 日志文件是否也在 containers 目录中呢？

我们进入/var/lib/docker/containers/<container_id>目录，就会看到一个名为<container_id>-json.log 的文件，它就是我们要寻找的 JSON 日志文件了，它的内容类似下面这样（为了方便阅读，此处对 JSON 数据进行了格式化）。

```
{
  "log": "180.166.202.194 - - [09/Mar/2017:03:32:03 +0000] \"GET /
HTTP/1.1\" 304 0 \"-\" \"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_3)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/56.0.2924.87 Safari/537.36\"
\"-\"\\n\",
  "stream": "stdout",
  "time": "2017-03-09T03:32:03.810882015Z"
}
```

原来如此，当我们通过 docker logs 命令所看到的日志，实际上就是解析这个 JSON 日志文件后的输出。

既然应用程序的日志可以写入 JSON 文件中，那么也能写入其他日志驱动中，json-file 只是 Docker 日志驱动的一种默认选项，Docker 已为我们提供了大量的日志驱动类型。

- none: 容器不输出任何日志；
- json-file: 容器输出的日志以 JSON 格式写入文件中（默认）；
- syslog: 容器输出的日志写入宿主机的 Syslog 中；
- journald: 容器输出的日志写入宿主机的 Journald 中；
- gelf: 容器输出的日志以 GELF（Graylog Extended Log Format）格式写入 Graylog 中；
- fluentd: 容器输出的日志写入宿主机的 Fluentd 中；
- awslogs: 容器输出的日志写入 Amazon CloudWatch Logs 中；

- splunk: 容器输出的日志写入 splunk 中;
- etwlogs: 容器输出的日志写入 ETW (Event Tracing for Windows) 中;
- gcplogs: 容器输出的日志写入 GCP (Google Cloud Platform) 中 ;
- nats: 容器输出的日志写入 NATS 服务器中。

我们可以在 `docker run` 命令中通过 `--log-driver` 参数来设置具体的 Docker 日志驱动, 也可以通过 `--log-opt` 参数来指定对应日志驱动的相关选项。就拿默认的 `json-file` 来说, 其实可以这样启动 Docker 容器:

```
docker run \  
-d \  
-p 80:80 \  
--log-driver json-file \  
--log-opt max-size=10m \  
--log-opt max-file=3 \  
--name nginx \  
nginx
```

我们通过 `--log-opt` 参数为 `json-file` 日志驱动添加了两个选项, `max-size=10m` 表示 JSON 文件最大为 10MB (超过 10MB 就会自动生成新文件), `max-file=3` 表示 JSON 文件最多为 3 个 (超过 3 个就会自动删除多余的旧文件)。

关于 Docker 日志驱动的更为详细的用法, 请参见 Docker 的官方文档。

Docker 日志驱动: <https://docs.docker.com/engine/admin/logging/overview/>。

在以上众多日志驱动类型中, 最为常用的是 Syslog, 因为 Syslog 是 Linux 的日志系统, 很多日志分析工具都可以从 Syslog 中获取日志, 比如流行的 ELK 日志中心, 它包括以下三个组件。

- (1) 日志存储: 由 Elasticsearch 负责。
- (2) 日志收集: 由 Logstash 负责。
- (3) 日志查询: 由 Kibana 负责。

在 ELK 的三个组件中, Logstash 用于收集日志, Syslog 中写入的日志可转发到 Logstash 中, 随后将日志存入 Elasticsearch 中, 最后可通过 Kibana 来查询日志。

我们在后面还会继续学习 ELK, 现在的主要任务是将 Docker 容器中输出的日志写入 Syslog, 那么后面需要做的就是将 Syslog 接入 ELK 了。

下面, 我们将关注的重点转移到 Syslog 上。

2.2.2 Linux 日志系统：Syslog

默认情况下，Linux 操作系统已安装了 Syslog 软件包，但它叫 Rsyslog。实际上，Rsyslog 是 Syslog 标准的一种实现。除了 Rsyslog 这一种实现，还有一种叫 Syslog-ng 的第三方实现。

Syslog-ng 官网：<https://www.balabit.com/network-security/syslog-ng>。

虽然 Syslog-ng 的功能较为强大，但我们还是选择使用 Rsyslog，因为操作系统已经预装了，我们无须再单独安装。当然，也可通过以下命令查看 Rsyslog 是否已安装。

```
rsyslogd -v
```

若输出以下内容，说明 Rsyslog 已安装，可正常使用。

```
rsyslogd 7.4.7, compiled with:
    FEATURE_REGEX:                               Yes
    FEATURE_LARGEFILE:                             No
    GSSAPI Kerberos 5 support:                     Yes
    FEATURE_DEBUG (debug build, slow code):        No
    32bit Atomic operations supported:              Yes
    64bit Atomic operations supported:              Yes
    Runtime Instrumentation (slow code):           No
    uuid support:                                   Yes
```

Rsyslog 的功能相当强大，我们可在它的官网中进一步学习。

Rsyslog 官网：<http://www.rsyslog.com>。

如果要开启 Rsyslog 服务，我们必须对 Rsyslog 进行配置，首先要做的就是打开它的配置文件：

```
vi /etc/rsyslog.conf
```

在 rsyslog.conf 文件中有一段配置，我们需要手工开启（去掉配置前面的注释，即“#”字符）。

```
$ModLoad imtcp
$InputTCPServerRun 514
```

开启以上配置，就能使用 TCP 协议连接 Rsyslog 的 514 端口（默认端口）。

Rsyslog 的配置文件修改完毕后，需手工重启 Rsyslog 服务，否则配置无法生效。

```
systemctl restart rsyslog
```

此时，我们可以查看一下本地是否对外开启了 514 端口。

```
netstat -anpt | grep 514
```

若出现类似下面的输出信息，则说明 Rsyslog 服务成功开启。

```
tcp    0  0.0.0.0:514  0.0.0.0:*  LISTEN  3721/rsyslogd
tcp6   0  :::514    :::*      LISTEN  3721/rsyslogd
```

随后可在启动 Docker 容器时连接 Rsyslog，从而将 Docker 容器输出的日志写入 Rsyslog 中。

通过以下命令启动 Nginx 容器，此时我们选择 Syslog 作为日志驱动。

```
docker run \
-d \
-p 80:80 \
--log-driver syslog \
--log-opt syslog-address=tcp://localhost:514 \
--name nginx \
nginx
```

首先需要使用--log-driver 指定 Syslog 为日志驱动，此外还需要使用--log-opt 指定 Syslog 服务器的地址 tcp://localhost:514，表示 Docker 容器可通过 TCP 协议连接本地的 514 端口，从而将日志写入 Rsyslog 中。

Nginx 容器启动成功后，我们同样在浏览器上发出请求，此时会发现，在 /var/lib/docker/containers/<container_id> 目录中不再生成 <container_id>-json.log 文件了。

那么日志输出到哪里去了呢？当然是在 Linux 的系统日志文件中了。此时我们可使用以下命令来查看 Linux 系统日志文件，该文件中的内容就是 Syslog 所生成的日志。

```
tail -f /var/log/messages
```

此时将看到类似下面的系统日志，手工刷新浏览器，可看到日志也在自动刷新。

```
Mar  9 15:59:11 localhost 67802e687e78[422]: 180.166.202.194 - -
```

```
[09/Mar/2017:07:59:11 +0000] "GET / HTTP/1.1" 304 0 "-" "Mozilla/5.0 (Macintosh;
Intel Mac OS X 10_12_3) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/56.0.2924.87 Safari/537.36" "-"
```

在以上日志信息中，我们可以看到一个 12 位的字符串 67802e687e78，其实它就是 Docker 的容器 ID。

我们知道，在一台宿主机上同时运行多个 Docker 容器是一件相当正常的事情。如果每个容器都通过 Syslog 来聚合日志，那么在系统日志文件中通过 Docker 容器的 ID 是很难识别出具体对应的是哪个容器的。能否在日志中添加容器的相关标识来区分某条日志来自于哪个容器呢？

其实 Docker 日志驱动早已为我们提供了支持，只需在 `--log-opt` 参数中添加一个 `tag` 选项，并在此选项上给出恰当的命名（可按容器或按功能命名），我们就能更好地识别出相应的日志。

```
docker run \
-d \
-p 80:80 \
--log-driver syslog \
--log-opt syslog-address=tcp://localhost:514 \
--log-opt tag="nginx" \
--name nginx \
nginx
```

将 `tag` 选项设置为 `nginx`（容器名称），就能在日志中看到带有 `nginx` 的标识，这样我们可以更加容易地识别这条日志来自 `Nginx` 容器。

```
Mar  9 16:33:54 localhost nginx[422]: 180.166.202.194 - -
[09/Mar/2017:08:33:54 +0000] "GET / HTTP/1.1" 304 0 "-" "Mozilla/5.0 (Macintosh;
Intel Mac OS X 10_12_3) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/56.0.2924.87 Safari/537.36" "-"
```

若不指定 `tag` 选项，则默认的 `tag` 为容器 ID 的前 12 个字符。我们也可在 `tag` 选项中使用 Docker 已提供的模板标签，可以将这些标签理解为 `tag` 选项中的占位符。

- `{{.ID}}`：容器 ID 的前 12 个字符；
- `{{.FullID}}`：容器 ID 的完整名称；
- `{{.Name}}`：容器名称；
- `{{.ImageID}}`：容器镜像 ID 的前 12 个字符；

- `{{.ImageFullID}}`: 容器镜像 ID 的完整名称;
- `{{.ImageName}}`: 容器镜像名称;
- `{{.DaemonName}}`: Docker 守护进程名称 (名为 docker)。

我们尝试使用几个模板标签, 试试最终的日志输出。

```
docker run \  
-d \  
-p 80:80 \  
--log-driver syslog \  
--log-opt syslog-address=tcp://localhost:514 \  
--log-opt tag="{{.ImageName}}/{{.Name}}/{{.ID}}" \  
--name nginx \  
nginx
```

启动容器后, 立即查看系统日志。

```
Mar  9 16:51:49 localhost nginx/nginx/6038c9789d01[422]: 180.166.202.194 -  
- [09/Mar/2017:08:51:49 +0000] "GET / HTTP/1.1" 304 0 "-" "Mozilla/5.0 (Macintosh;  
Intel Mac OS X 10_12_3) AppleWebKit/537.36 (KHTML, like Gecko)  
Chrome/56.0.2924.87 Safari/537.36" "-"
```

可见, 日志中所出现的 `nginx/nginx/6038c9789d01` 正是我们在 `tag` 选项中指定的 `{{.ImageName}}/{{.Name}}/{{.ID}}`。

最后, 我们通过一幅架构图, 总结一下本节所涉及的 Docker 日志驱动内容。

2.2.3 Docker 日志架构

Docker 容器 (Docker Container) 中的应用程序 (Application) 将日志写入到标准输出设备 (STDOUT), Docker 守护进程 (Docker Daemon) 负责从 STDOUT 中获取日志, 并将日志写入对应的日志驱动中, 如图 2-1 所示。

目前, 应用程序中的日志已经从 Docker 容器内部成功写入宿主机的 Syslog 中, 接下来我们要做的是, 将 Syslog 中的日志转发到 ELK 平台的 Logstash 中, 从而建立我们所需要的“应用日志中心”, 如图 2-2 所示。

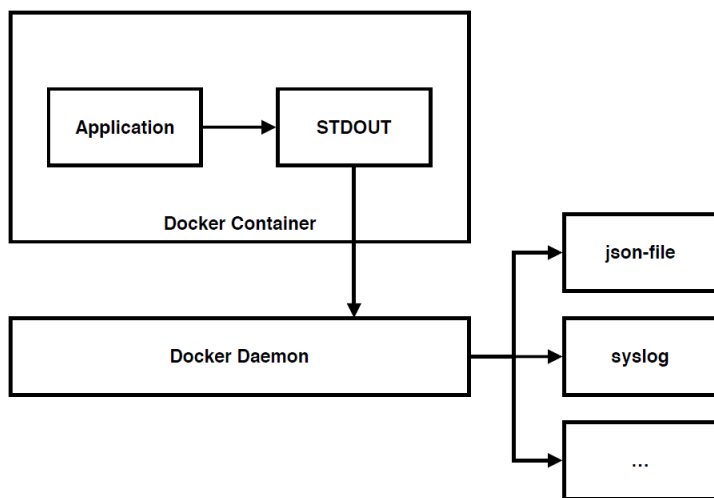


图 2-1 Docker 日志架构

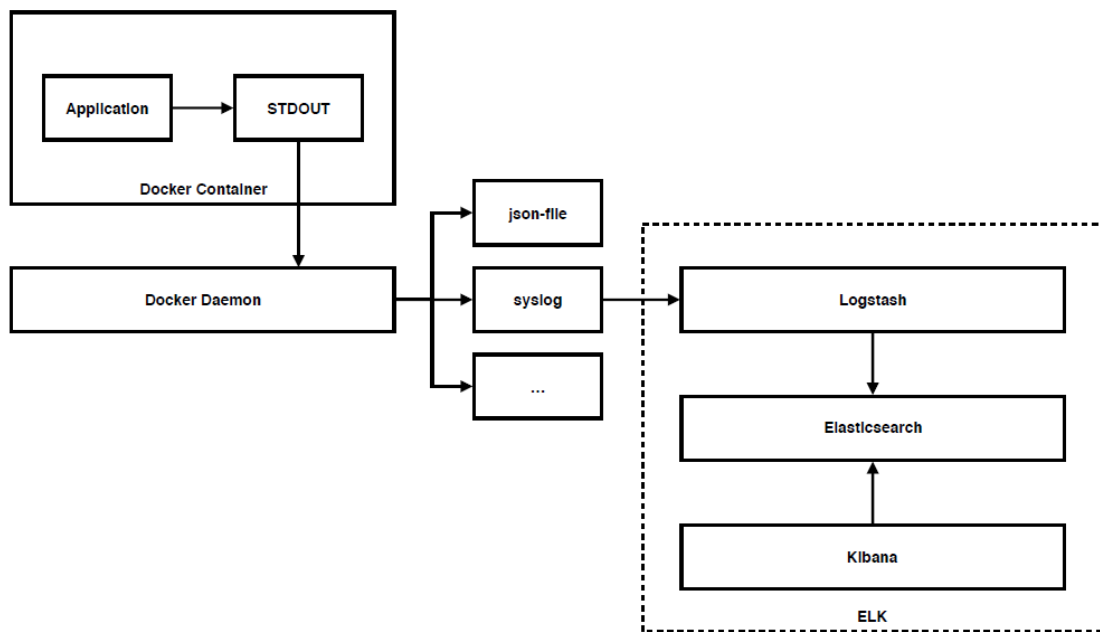


图 2-2 应用日志中心架构

以上架构便是我们需要建设的目标，接下来，我们进入 ELK 的世界。

2.3 搭建应用日志中心

我们想要搭建的应用日志中心，主要是为了收集应用程序所产生的日志，并通过这个平台来存储这些日志，最终以可视化的方式对日志进行查询与分析。业界流行的一款开源的日志中心，它的功能相当强大，产品化程度非常高，不管是用户界面还是参考文档，它都在不断地追求更好的品质，这款开源日志中心就是 ELK。实际上，ELK 是由三个开源产品组成的，即 Elasticsearch、Logstash、Kibana，这三个开源产品归属于一家叫 Elastic 的公司。

下面我们就来介绍一下 Elastic 公司的核心产品 ELK。

2.3.1 开源日志中心：ELK

从 Elastic 公司的官网上，我们就能快速了解该公司所提供的产品，如图 2-3 所示。用官方的说法，叫“开源 Elastic 栈（Open Source Elastic Stack）”。

Elastic 官网：<https://www.elastic.co/>。

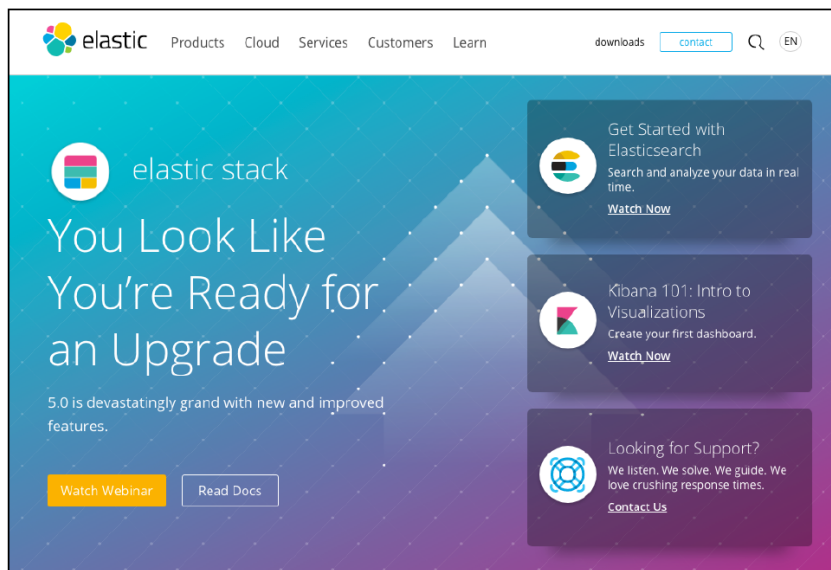


图 2-3 Elastic 官网

从官网上可知，Elastic 官方推出了 6 款开源产品。

- （1）Kibana：用于数据可视化。
- （2）Elasticsearch：用于数据搜索、分析与存储。

- (3) Logstash: 用于数据收集, 将数据存入 Elasticsearch 中。
- (4) Beats: 用于数据传输, 将数据从磁盘上传输到 Logstash 中。
- (5) X-Pack: 提供一些扩展功能, 包括安全、预警、监控、报表、图形化等。
- (6) Elastic Cloud: 提供 Elastic 栈的云服务, 提供公有云与私有云解决方案。

ELK 是 Elastic 栈中的核心技术, 我们有必要逐个学习, 不妨先来了解一下 Elasticsearch, 因为它是整个日志中心的数据存储中心。

2.3.2 日志存储系统: Elasticsearch

Elasticsearch 是一个可高度扩展的开源全文搜索与分析引擎, 它可以帮助我们快速地存储、搜索与分析大规模的实时数据。Elasticsearch 的底层基于开源搜索引擎 Lucene, 并在此基础上提供了一系列便于应用程序使用的 REST API, 并且还提供了先天性的集群能力, 可自由水平扩展以支持日益增长的数据。

我们可从 Elasticsearch 的官网了解一下它的基本功能。

Elasticsearch 官网: <https://www.elastic.co/products/elasticsearch>。

在学习 Elasticsearch 之前, 我们必须先搞清楚它的几个基本概念。

实际上, Elasticsearch 所提供的是一个接近于实时的搜索能力, 我们称其为“近实时”(Near Realtime, NRT)。由于 Elasticsearch 的底层用到了 Lucene, 当数据进入 Elasticsearch 后, 首先要进行索引 (Index), 这个过程需要花费一段非常短暂的延迟时间。当我们进行查询的时候, 需要通过索引来检索真正的数据, 因此会产生一个延迟, 官方宣称这个时间大约需要 1 秒。

我们可以把 Elasticsearch 中的索引想象成关系型数据库中的数据表, 表中的每行记录对应着 Elasticsearch 中的每篇文档 (Document), 与此不同的是, Elasticsearch 在索引和文档之间还提供了一个类型 (Type) 的概念, 它用于表示每篇文档所对应的数据类型。

Elasticsearch 具备先天性的集群能力, 其实在最早设计的时候, 集群特性已经被考虑进来。在大多数拥有集群特性的技术中, 都会提到三个概念: 节点 (Node)、分片 (Shards)、副本 (Replicas), 毫无疑问, Elasticsearch 集群也不例外。在集群环境中, 每个 Elasticsearch 实例就是一个节点。由于整个集群存储了巨大的数据量, 即使能做到将整个数据存储于单个节点中, 但却无法保证该节点的计算能力达到所期望的要求, 因此需要对节点上的数据进行分片。也就是说, 整个数据在集群中被划分为多个片段, 每个片段被存放在不同的节点上。由于分布式环境中可能运行了多个节点, 此外分布式环境可能出现无法避免的故障, 因此每个节点中的分片数据需要做副本, 以避免因故障导致整个集群出现数据丢失。我们可在 Elasticsearch 集群环境

中，根据实际情况来设置节点、分片、副本的数量。

对 Elasticsearch 有了一些了解以后，我们不妨亲自动手体验一下它的基本使用方法。

最简单的安装方式莫过于 Docker 容器，我们可使用以下 `docker run` 命令来启动 Elasticsearch 容器。

```
docker run \
--rm \
-p 9200:9200 \
--name elasticsearch \
elasticsearch
```

此时我们将 Elasticsearch 的默认 HTTP 端口号（9200）暴露到容器外部，以便可通过浏览器或应用程序进行访问。

容器启动完毕后，我们可在浏览器中输入 `http://localhost:9200/` 来访问 Elasticsearch。

```
{
  "name" : "tVZvo0x",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "ay3gs6zaRciFWV6_A9pG1Q",
  "version" : {
    "number" : "5.2.2",
    "build_hash" : "f9d9b74",
    "build_date" : "2017-02-24T17:26:45.835Z",
    "build_snapshot" : false,
    "lucene_version" : "6.4.1"
  },
  "tagline" : "You Know, for Search"
}
```

这里显示了 Elasticsearch 节点的名称（`name`），还有集群名称（`cluster_name`）与集群 UUID（`cluster_uuid`）。此外，还包括 Elasticsearch 的版本信息，包括版本号、构建信息、Lucene 版本号等。最后，还提供了一个看似不太重要的标签行（`tagline`）。

在启动 Elasticsearch 时，我们可传入一些启动参数来配置集群信息，就像下面这样。

```
docker run \
...
elasticsearch -Ecluster.name=my_cluster_name -Enode.name=my_node_name
```

我们可通过-E 参数指定集群名称（cluster.name）与节点名称（node.name）。

需要注意的是，若要开启 Elasticsearch 的集群特性，则还需对外包括 9300 端口，它是集群节点之间进行 TCP 通信的端口号。

我们通过以上方式，成功启动了 Elasticsearch 集群中的一个节点，也可使用同样的方式来启动更多的节点。但我们仍然希望先在单节点上进行快速入门，后面再不断深入学习。

最容易判断 Elasticsearch 集群是否可用的方式就是，打开浏览器，在地址上输入 `http://localhost:9200/_cat/health?v`，随后会看到两行文字，用于描述 Elasticsearch 集群的健康情况，也就是说，判断该集群是否可用。

```
epoch          timestamp cluster          status node.total node.data shards pri
relo init unassign pending_tasks max_task_wait_time active_shards_percent
1489385818 06:16:58 elasticsearch
green          1          1          0  0  0  0          0          0  -      100.0%
```

上面的信息实际上是一张表格，第一行相当于一个表头（字段名），第二行是该表格中的数据，表示该 Elasticsearch 集群中包含一个节点。

如果我们对以上表格中每个表头的意义不太明确，可将?v 参数改为?help，此时会看到所有表头的含义。

Epoch	t,time	seconds since 1970-01-01 00:00:00
timestamp	ts,hms,hmmss	time in HH:MM:SS
cluster	cl	cluster name
status	st	health status
node.total	nt,nodeTotal	total number of nodes
node.data	nd,nodeData	number of nodes that can store data
shards	t,sh,shards.total, shardsTotal	total number of shards
pri	p,shards.primary, shardsPrimary	number of primary shards
relo	r,shards.relocating, shardsRelocating	number of relocating nodes
init	i,shards.initializing, shardsInitializing	number of initializing nodes

续表

unassign	u,shards.unassigned, shardsUnassigned	number of unassigned shards
pending_tasks	pt,pendingTasks	number of pending tasks
max_task_wait_time	mtwt,maxTaskWaitTime	wait time of longest task pending
active_shards_percent	asp,activeShardsPercent	active number of shards in percent

如果我们希望以 JSON 格式输出数据，可以使用?format=json；如果想要看到格式化效果的 JSON 数据，可在参数后添加&pretty 参数，即 ?format=json&pretty。

```
[
  {
    "epoch" : "1489385968",
    "timestamp" : "06:19:28",
    "cluster" : "elasticsearch",
    "status" : "green",
    "node.total" : "1",
    "node.data" : "1",
    "shards" : "0",
    "pri" : "0",
    "relo" : "0",
    "init" : "0",
    "unassign" : "0",
    "pending_tasks" : "0",
    "max_task_wait_time" : "-",
    "active_shards_percent" : "100.0%"
  }
]
```

以上查看集群健康情况的特性是由 Elasticsearch 的 CAT API 提供的，它是一个 REST API，我们可添加指定的参数来得到想要的输出结果。

- 显示详情模式：?v;
- 查看字段描述：?help;
- 指定表头字段：?h;

- 制定显示单位: ?bytes=b;
- 指定输出格式: ?json;
- 美化格式输出: ?format=json&pretty;
- 指定排序方式: ?v&s=foo:asc,bar:desc。

关于 CAT API 更为详细的介绍, 请参见 CAT API 文档。

CAT API 文档:<https://www.elastic.co/guide/en/elasticsearch/reference/current/cat.html>

使用 CAT API 除了可以查看集群健康情况, 还能查看集群节点信息。我们只需在浏览器中输入 `http://localhost:9200/_cat/nodes?v`, 就能看到该 Elasticsearch 集群中所有节点的具体信息。

```

ip          heap.percent ram.percent cpu load_1m load_5m load_15m node.role
master name
localhost   19          33    0    0.00    0.01    0.05
mdi        *          tVZvo0x

```

当然, 我们也可使用 `curl` 或其他工具调用 CAT API 来获取以上信息, 此时使用浏览器只是为了便于操作。

Elasticsearch 除了具备强大的数据存储与计算能力, 也提供了类似 CAT API 这样易用的 REST API, 不仅利于应用程序调用, 还便于开发者使用。对于开发者而言, 使用浏览器来调用 REST API 是有局限性的, 由于浏览器仅便于发送 GET 请求, 却不能发送 REST API 所包含的其他请求 (POST、PUT、DELETE), 因此我们不得不寻找更好用的 RSET API 客户端工具。`curl` 是一种不错的选择, 我们可在命令行中执行 `curl` 命令来发送 REST API 请求, 但操作过程却没有图形化支持。幸运的是, 我们找到了比 `curl` 更好用的利器, 它就是 Postman。

Postman 实际上是 Chrome 浏览器的一个插件, 也可以单独安装。它是一款图形化界面的 REST API 客户端工具, 安装后几乎不用任何的学习成本, 就能快速学会它的使用方法。

Postman 官网: <https://www.getpostman.com/>。

下面我们就用 Postman 来体验一下 Elasticsearch 强大的 REST API。

1) 创建索引

首先, 我们使用以下 PUT 请求, 创建一个名为 `customer` 的索引。

```
PUT http://localhost:9200/customer
```

在 Postman 左上角的下拉框中选择 PUT 类型，并输入 `http://localhost:9200/customer` 地址，敲击“回车”或按下“Send”按钮，即可发送 REST API 请求。

发送 REST API 请求后，可看到如下输出：

```
{
  "acknowledged": true,
  "shards_acknowledged": true
}
```

该输出表示索引创建完毕，节点已收到确认（`acknowledged`），分片也已确认（`shards_acknowledged`）。

随后，我们可使用 CAT API 查看 Elasticsearch 的索引信息，只需在浏览器中输入 `http://localhost:9200/_cat/indices?v` 即可。

health	status	index	uuid		pri	rep	docs.count
docs.deleted	store.size	pri.store.size					
yellow	open	customer					
oR6-gUwdTkGA96ReQ26MDA	5	1	0	0	650b	650b	

以上信息表示索引已创建成功，但该索引中尚未包含任何文档（`docs.count` 为 0），随后我们来创建一条文档。

2) 创建文档

使用以下 PUT 请求，我们在 `customer` 索引中创建一条 `external` 类型 ID 为 1 的文档，为了便于描述，下文简称 `/customer/external/1` 文档。

```
PUT http://localhost:9200/customer/external/1
```

此时，需要在 Body 的 raw 中添加以下 JSON 数据：

```
{
  "name": "John Doe"
}
```

发送 REST API 请求后，可看到如下输出：

```
{
  "_index": "customer",
```



```
{
  "_type": "external",
  "_id": "1",
  "_version": 1,
  "result": "created",
  "_shards": {
    "total": 2,
    "successful": 1,
    "failed": 0
  },
  "created": true
}
```

以上信息表示文档已创建成功，该文档对应的索引是 `customer`，类型 `external`，ID 是 1。

如果我们再次发送同样的 REST API 请求，但是 ID 保持相同，只是内容稍加改变，那么此时将不再创建新的文档，而是根据 ID 修改已有文档，该特性是 Elasticsearch 所提供的。

现在已经创建了一条文档，下面我们就将此文档查询出来。

3) 查询文档

使用以下 GET 请求来查询 `/customer/external/1` 文档。

```
GET http://localhost:9200/customer/external/1
```

发送 REST API 请求后，可看到如下输出：

```
{
  "_index": "customer",
  "_type": "external",
  "_id": "1",
  "_version": 1,
  "found": true,
  "_source": {
    "name": "John Doe"
  }
}
```

在返回的 JSON 数据中，有一个 `_source` 的字段，它就是文档的实际数据，下面我们尝试修改这份数据。

4) 修改文档

使用以下 POST 请求，修改/customer/external/1 文档。

```
POST http://localhost:9200/customer/external/1
```

此时，我们将 Body 数据修改为以下内容，将 John Doe 修改为 Jane Doe。

```
{
  "name": "Jane Doe"
}
```

发送 REST API 请求后，可看到如下输出。

```
{
  "_index": "customer",
  "_type": "external",
  "_id": "1",
  "_version": 2,
  "result": "updated",
  "_shards": {
    "total": 2,
    "successful": 1,
    "failed": 0
  },
  "created": false
}
```

以上输出表示操作成功，此时该文档的版本号增长为 2（以前是 1）。

除了以上修改文档的方法，还有另一种方法也能修改文档。同样是 POST 请求，只是 URL 与 Body 有所不同，此时的 URL 中带有_update 后缀。

```
POST http://localhost:9200/customer/external/1/_update
```

Body 中是一个固定格式的 JSON 数据，被修改的数据统一放在 doc 字段中。

```
{
  "doc": {
    "name": "Jane Doe"
  }
}
```

当然也可以在已有文档中添加新的字段，此时我们再添加一个 `age` 字段。

```
{
  "doc": {
    "name": "Jane Doe",
    "age": 20
  }
}
```

我们还能修改文档中相应的字段，比如将 `age` 字段增加 5。

```
{
  "script": "ctx._source.age += 5"
}
```

其中，`ctx` 表示文档上下文对象，`_source` 表示文档内容。

我们现在学会了创建文档和修改文档的方法，再来看看如何删除文档。

5) 删除文档

使用以下 POST 请求来删除 `/customer/external/1` 文档。

```
DELETE http://localhost:9200/customer/external/1
```

发送 REST API 请求后，可看到如下输出：

```
{
  "found": false,
  "_index": "customer",
  "_type": "external",
  "_id": "1",
  "_version": 1,
  "result": "not_found",
  "_shards": {
    "total": 2,
    "successful": 1,
    "failed": 0
  }
}
```

以上输出表示操作成功，已找不到该文档。

既然可以删除文档，那么同样也可以删除索引。

6) 删除索引

使用以下 POST 请求来删除 `customer` 索引。

```
DELETE http://localhost:9200/customer
```

发送 REST API 请求后，可看到如下输出。

```
{
  "acknowledged": true
}
```

以上输出表示操作成功，索引已不存在。

此时我们已学会了 Elasticsearch 的常用管理功能，实际上更为常用的功能是查询文档，Elasticsearch 提供了强大的 Search API。可通过以下文档，了解 Search API 的使用方法。

Search API 文档: <https://www.elastic.co/guide/en/elasticsearch/reference/current/search.html>。

更多关于 Elasticsearch 的使用方法，请参见以下文档。

Elasticsearch 文档: <https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html>。

我们要搭建的是一个应用日志中心，首先日志可以通过 Logstash 来收集，然后将收集到的日志存入 Elasticsearch 中，最后通过 Kibana 来查询日志。

下面，我们来学习 Logstash 日志收集系统。

2.3.3 日志收集系统：Logstash

Logstash 是一款开源的数据收集引擎，它既提供了实时管道能力，也提供了灵活的插件机制，我们可以自由选择已有的插件，也能自行开发所需的插件。我们使用 Logstash 更多的时候都是在做参数配置，以实现我们所需的功能。

我们可从 Logstash 的官网了解一下它的基本功能。

Logstash 官网: <https://www.elastic.co/products/logstash>。

我们从系统架构的角度来看 Logstash，实际上它提供了三个内部组件，分别是输入组件（INPUTS）、过滤组件（FILTERS）、输出组件（OUTPUTS），而且每个组件都提供了插件机制。我们可以将这些组件及其插件想象为一个管道（Pipeline），数据从数据源（Data Source）流向 INPUTS、FILTERS、OUTPUT，最终到达 Elasticsearch 中进行存储，如图 2-4 所示。

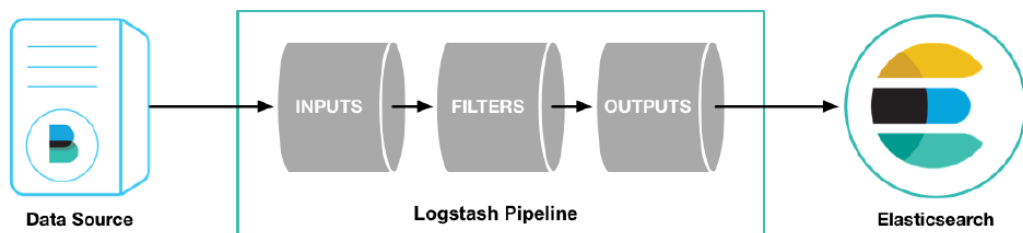


图 2-4 Logstash 系统架构

对于 INPUTS 与 OUTPUTS 而言，还提供了编解码（Codec）机制，可以更安全地传输数据。针对以上提到的三个组件，我们简单介绍一下它们所包含的常用插件。

1) 输入组件

- file: 读取文本文件。
- syslog: 读取 Syslog（包含 Rsyslog）。
- redis: 读取 Redis 消息队列。
- beats: 处理 Filebeat 事件。

2) 过滤插件

- grok: 解析日志文本。
- mutate: 修改事件字段。
- drop: 删除事件。
- clone: 复制事件。
- geoip: 添加 IP 地理位置。

3) 输出插件

- elasticsearch: 写入 Elasticsearch。
- file: 写入文本文件。
- graphite: 写入 Graphite（一款开源工具，用于存储与图形化指标）。
- statsd: 写入 statsd（统计服务，监听 UDP 端口）。

4) 编解码插件

- `json`: 编解码为 JSON 格式。
- `multiline`: 合并多行文本（例如，合并多行 Java 异常信息为单行文本）。

以上仅为 Logstash 的常用插件，可从以下链接中了解它的所有插件。

Logstash 输入插件: <https://www.elastic.co/guide/en/logstash/current/input-plugins.html>。

Logstash 过滤插件: <https://www.elastic.co/guide/en/logstash/current/filter-plugins.html>。

Logstash 输出插件: <https://www.elastic.co/guide/en/logstash/current/output-plugins.html>。

Logstash 编解码插件: <https://www.elastic.co/guide/en/logstash/current/codec-plugins.html>。

当我们简单了解了 Logstash 的架构与功能以后，有必要亲自动手来体验一下。

最容易运行 Logstash 的方式还是 Docker 容器，我们可通过以下 `docker run` 命令来启动 Logstash 容器。

```
docker run \
-it \
--rm \
logstash \
logstash -e 'input { stdin { } } output { stdout { } }'
```

我们通过交互方式（`-it`）启动了 Docker 容器，并指定退出容器后可立即删除容器（`--rm`），进入容器后需要运行一句 `logstash` 命令，并将一段字符串参数传入该命令（`-e`）。该字符串实际上是 Logstash 的配置，表示从标准输入设备（`stdin`）中获取字符，并将字符输出到标准输出设备（`stdout`）中。

如果 Logstash 容器启动成功，将在控制台中输出以下内容：

```
Settings: Default pipeline workers: 8
Pipeline main started
```

表示 Logstash 默认的管道进程（`pipeline workers`）有 8 个，它实际上为 CPU 的核心数。

如果大家没有看到以上输出，也许是因为使用了当前最新的 5.2.2 版本。经我们不断尝试，发现该版本对应的 Docker 镜像无法正常使用，我们只能选择较为稳定的 2.4.1 版本。此时，需

要将 Docker 镜像名称改为 `logstash:2.4.1`。

此时，我们可在键盘上输入任意字符，比如 `hello`，敲下回车键后就能在控制台中看到一行日志。

```
hello
2017-03-10T02:26:23.677Z 3b5c1ab80725 hello
```

第一段（`2017-03-10T02:26:23.677Z`）表示日志的时间戳，第二段（`3b5c1ab80725`）表示 Logstash 容器的 ID，第三段（`hello`）表示日志的消息内容。

在启动 Logstash 容器时，我们通过 `logstash` 命令行的 `-e` 参数传入了一段字符串，对于目前简单体验的情况来说，是可以这样操作的，但在实际过程中，会有更复杂的配置。我们一般会将该字符串写入配置文件中，并在启动 Logstash 容器时传入该配置文件。

我们不妨在宿主机上创建一个 Logstash 配置文件，该文件可存放在容易访问的目录中。

```
mkdir ~/logstash
vi ~/logstash/logstash.conf
```

将以上字符串复制到 `logstash.conf` 配置文件中，为了方便阅读和维护，我们将其内容进行了格式化。

```
input {
  stdin {
  }
}
output {
  stdout {
  }
}
```

当 Logstash 配置文件准备完毕后，我们就能使用以下方式来启动 Logstash 容器了。

```
docker run \
-it \
--rm \
-v ~/logstash/logstash.conf:/etc/logstash.conf \
--name logstash \
logstash \
logstash -f /etc/logstash.conf
```

由于 `logstash.conf` 配置文件在宿主机上，我们需要将该文件通过数据卷的方式映射到容器内部（`-v`），并在执行 `logstash` 命令后传入该配置文件（`-f`）。

当然，我们也可以使用 `Dockerfile` 来制作 `Logstash` 镜像，将配置文件写入镜像中，启动容器时就无须传入任何参数了。

在 `logstash.conf` 文件所在的目录中创建一个名为 `Dockerfile` 的文件，内容如下：

```
FROM logstash:latest
COPY logstash.conf /etc/
CMD ["logstash", "-f", "/etc/logstash.conf"]
```

首先通过 `FROM` 指令指定该镜像继承于 `logstash:latest` 镜像，然后通过 `COPY` 指令将当前目录中的 `logstash.conf` 文件复制到镜像的 `/etc/` 目录中，最后通过 `CMD` 指令来指定进入容器后需要执行的命令。

现在需要做的是，在 `Dockerfile` 文件所在的目录中执行 `docker build` 命令，从而生成我们所需的 `Logstash` 镜像。

```
docker build -t huangyong/logstash .
```

我们通过 `-t` 参数来指定镜像名称为 `huangyong/logstash`，`Dockerfile` 文件的路径就在当前目录中，用 `“.”` 表示。

可通过 `docker images` 命令来查看是否成功生成了 `Logstash` 镜像，如果该镜像已存在，就可以通过以下 `docker run` 命令来启动 `Logstash` 容器。

```
docker run \
-it \
--rm \
--name logstash \
huangyong/logstash
```

此外，当我们在控制台上输入字符，并敲下回车键后，看到的是一行日志，如果我们想让输出的日志更加结构化一些，可在 `stdout` 插件中使用 `codec` 操作，对需要输出的日志进行格式化。

```
output {
  stdout {
    codec => rubydebug
  }
}
```



```

    }
  }

```

以上表示使用 `rubydebug` 的方式对日志进行格式化，这样我们就能在控制台中看到一个 Ruby 对象的日志了。

```

{
  "message" => "hello",
  "@version" => "1",
  "@timestamp" => "2017-03-10T06:01:09.346Z",
  "host" => "30cc676f0c74"
}

```

我们很清晰地发现，该日志包含四个字段。

- (1) `message`: 日志消息。
- (2) `@version`: 日志版本（自动增长的）。
- (3) `@timestamp`: 日志时间（精确到毫秒）。
- (4) `host`: 日志域名（与容器 ID 相同）。

带有“@”前缀的字段是 Logstash 所提供的系统字段，我们也可以在 `stdin` 插件中进行 `add_field` 操作，添加一些自定义字段，这些字段也会出现在日志对象中。

```

input {
  stdin {
    add_field => {
      "field1" => "foo"
      "field2" => "bar"
    }
  }
}

```

重新启动容器，在输出日志内容中就会看到我们已添加的字段。

```

{
  "message" => "hello",
  "@version" => "1",
  "@timestamp" => "2017-03-10T06:25:08.258Z",
  "field1" => "foo",
  "field2" => "bar",
}

```

```
    "host" => "3ec7dd362a5d"
  }
```

我们还可以使用 `filter` 组件，在该组件中使用 `mutate` 来移除指定的字段。

```
input {
  ...
}
filter {
  mutate {
    remove_field => [ "field2" ]
  }
}
output {
  ...
}
```

重新启动容器，在输出日志内容中看不到我们已移除的字段。

```
{
  "message" => "hello",
  "@version" => "1",
  "@timestamp" => "2017-03-10T06:27:27.195Z",
  "field1" => "foo",
  "host" => "2c0202fab125"
}
```

Logstash 的功能相当强大，以上仅仅是为了快速上手，在实际应用中我们还需要深入学习。

Logstash 文档: <https://www.elastic.co/guide/en/logstash/current/index.html>。

最后，我们来学习 Kibana 日志查询系统。

2.3.4 日志查询系统：Kibana

Kibana 是一个开源的基于 Elasticsearch 的分析与可视化平台，我们既可用它来查看并搜索存储在 Elasticsearch 中的数据，也可用它来制作各式各样的图表、表格、地图等图形化数据。

我们可通过 Kibana 的官网来快速了解它的核心功能与使用方法。

Kibana 官网: <https://www.elastic.co/products/kibana>。

同样, 我们也可使用 Docker 容器来运行 Kibana。

```
docker run \
--rm \
-p 5601:5601 \
--link elasticsearch:elasticsearch \
-e ELASTICSEARCH_URL=http://elasticsearch:9200 \
--name kibana \
kibana
```

我们对外暴露了 5601 端口, 它是 Kibana 默认对外提供的 HTTP 端口, 我们可在浏览器中访问 Kibana。我们还将 Kibana 容器连接到 Elasticsearch 容器上(--link), 此时需要添加一个 Docker 环境变量 (-e), 它是 Kibana 连接 Elasticsearch 所需的 URL (ELASTICSEARCH_URL)。

Kibana 容器启动完毕后, 打开浏览器, 输入 <http://localhost:5601/>, 此时会看到 Kibana 的索引配置界面, 如图 2-5 所示。

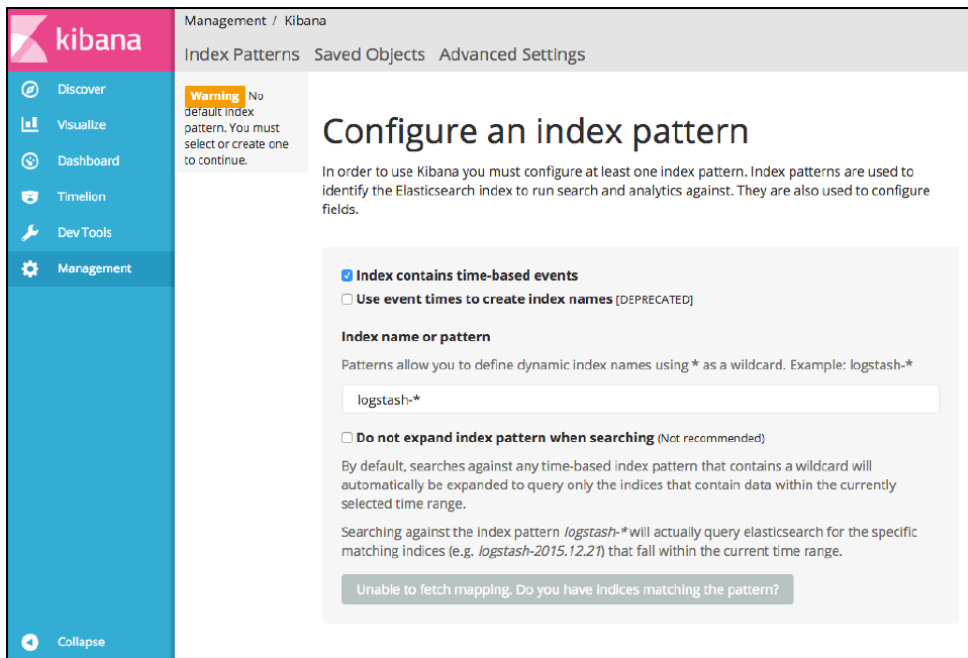


图 2-5 Kibana 索引配置界面

此时, Kibana 提示我们配置 Logstash 所产生的索引名称, 默认是 `logstash-*`, 表示 Kibana

可从 Elasticsearch 中读取以 “logstash-” 开头的索引。

虽然 Kibana 已与 Elasticsearch 集成，但 Elasticsearch 中尚未存入任何日志，因此 Kibana 目前无法使用。

更多关于 Kibana 的使用方法，请参见以下文档。

Kibana 文档：<https://www.elastic.co/guide/en/kibana/current/index.html>。

在熟悉了 Elasticsearch、Logstash、Kibana 这三个系统的基本用法以后，我们一起动手搭建 ELK 日志中心。

2.3.5 搭建 ELK 日志中心

在我们开始搭建 ELK 日志中心之前，需要首先理解 ELK 三者之间的关系以及数据流向，因为这些知识决定了我们先启动哪个容器，后启动哪个容器。由于 Elasticsearch 是整个 ELK 平台的日志存储系统，因此我们需要先启动 Elasticsearch 容器。随后可启动 Rsyslog 服务与 Logstash 容器，它们共同完成日志的收集工作。最后启动 Kibana 容器，它从 Elasticsearch 容器中查询日志。当然，我们也需要借用一个应用程序容器来产生日志，不妨以 Nginx 为例，它的日志将通过 Rsyslog 流入 Logstash，然后存入 Elasticsearch，最后在 Kibana 中查询。

1) 启动 Elasticsearch 容器

我们通过以下 docker run 命令来启动 Elasticsearch 容器。

```
docker run \
-d \
-p 9200:9200 \
-v ~/elasticsearch/data:/usr/share/elasticsearch/data \
--name elasticsearch \
elasticsearch
```

此时我们将此容器在后台运行（-d），并将 Elasticsearch 的数据目录映射到宿主机上（-v），这样可有助于我们备份 Elasticsearch 所产生的数据。

可在浏览器上通过访问 <http://47.93.82.56:9200/> 来查看 Elasticsearch 是否可用。

如果 Elasticsearch 日志存储系统已经就绪，那么我们就来启动 Logstash 日志收集系统。

2) 启动 Logstash 容器

在启动 Logstash 容器之前，我们需要先配置 Rsyslog，将其收集到的日志通过 TCP 端口转

发的方式传输到 Logstash 中。

首先我们打开 Rsyslog 的配置文件/etc/rsyslog.conf，开启以下配置：

```
$ModLoad imtcp
$InputTCPServerRun 514

*. * @@localhost:4560
```

前两行配置表示 Rsyslog 加载 imtcp 模块并监听 514 端口，最后一行配置表示将 Rsyslog 中收集的数据转发到本地的 4560 端口上。这个 4560 端口就是 Logstash 输入组件的端口，后面我们会提到。

Rsyslog 的配置文件修改完毕后，我们需要手工重启服务，才能使配置生效。

```
systemctl restart rsyslog
```

随后我们打开 Logstash 的配置文件~/logstash/logstash.conf，在 input 与 output 组件中添加相关插件。

```
input {
  syslog {
    type => "rsyslog"
    port => 4560
  }
}
output {
  elasticsearch {
    hosts => [ "elasticsearch:9200" ]
  }
}
```

在 input 组件中，我们添加了 syslog 插件，它包含 type 与 port 两个参数。type 指定为 rsyslog，表示对接 Rsyslog 服务。port 设置为 4560，即为 Rsyslog 的 TCP 转发端口。在 output 组件中，我们添加了 elasticsearch 插件，只需指定 Elasticsearch 的域名与端口即可，此处也可指定 Elasticsearch 集群的多个节点，只需在数组中用逗号分隔即可。

配置完成后，我们通过以下 docker run 命令来启动 Logstash 容器。

```
docker run \
-d \
```

```
-p 4560:4560 \  
-v ~/logstash/logstash.conf:/etc/logstash.conf \  
--link elasticsearch:elasticsearch \  
--name logstash \  
logstash \  
logstash -f /etc/logstash.conf
```

我们对外暴露了 Logstash 需要对外提供的 4560 端口（-p），并挂载了自定义的 Logstash 配置文件（-v）。由于 Logstash 输出的日志需要写入 Elasticsearch 中，因此需要将 Logstash 容器连接到 Elasticsearch 容器上（--link）。在运行 logstash 命令时，需要指定映射到容器内部的配置文件。

现在日志收集系统已经准备完毕，下面我们就来启动 Kibana 日志查询系统。

3) 启动 Kibana 容器

我们通过以下 docker run 命令来启动 Kibana 容器。

```
docker run \  
-d \  
-p 5601:5601 \  
--link elasticsearch:elasticsearch \  
-e ELASTICSEARCH_URL=http://elasticsearch:9200 \  
--name kibana \  
kibana
```

由于 Kibana 需要从 Elasticsearch 中读取数据，因此需要将 Kibana 容器连接到 Elasticsearch 容器上（--link），此时还需通过环境变量的方式（-e）将 Elasticsearch 连接配置（ELASTICSEARCH_URL）传入 Docker 容器内部，以使 Kibana 可从环境变量中获取 Elasticsearch 的连接配置。

现在 ELK 平台已搭建完毕，可通过访问 <http://localhost:5601/> 来进入 Kibana。

此时 Kibana 的 Discover 面板中尚无任何日志，因为我们接下来就要启动一个 Nginx 容器，将其作为生成日志的来源。

4) 启动 Nginx 容器（生产日志）

我们通过以下 docker run 命令来启动 Nginx 容器。

```
docker run \  
-d \  
-p 80:80 \  
nginx
```

```
--log-driver syslog \
--log-opt syslog-address=tcp://localhost:514 \
--log-opt tag="nginx" \
--name nginx \
nginx
```

我们将日志驱动设置为 `syslog` (`--log-driver`)，传入 `Rsyslog` 的连接配置 (`--log-opt syslog-address`)，并将日志标签 (`--log-opt tag`) 设置为 `nginx`，这样有助于在 `Logstash` 中快速识别出 `Nginx` 的日志。

`Nginx` 容器启动完毕后，我们打开浏览器，访问 `http://localhost/`，就能看到 `Nginx` 首页，在浏览器中手工刷新几下页面，就能在 `Kibana` 的 `Discover` 面板中看到相应的日志输出了，如图 2-6 所示。

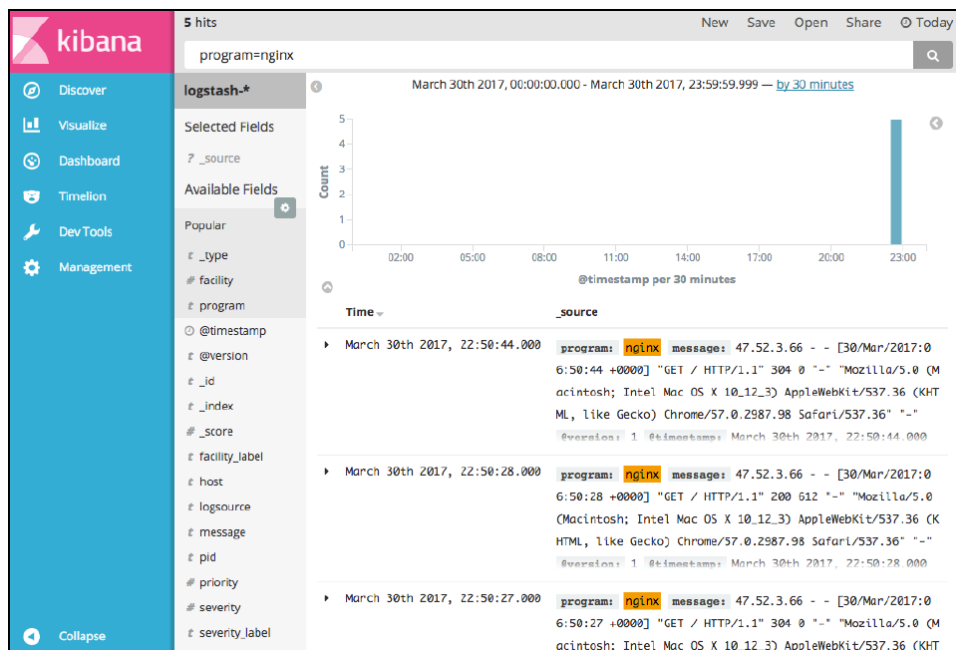


图 2-6 Kibana 日志查询

可在右上角的 `Time Rang` 中将时间设置为 `Today`，可查看当天的所有日志。此时可能会看到其他系统日志，可在最上方的输入框中填入 `program=nginx` 作为查询条件，点击查询按钮，将过滤出 `Nginx` 的相关日志。

此时我们发现，现在是 15 点，但日志时间却是 23 点，也就是说，日志时间比现在时间晚 8 小时，很明显这是 `Kibana` 的时区问题导致的。为了解决这个问题，我们需要在 `Kibana`

中进行设置，将时区设置为 UTC。操作方法是：进入 Management 面板，点击 Advanced Settings 链接，找到 `dateFormat:tz` 参数，在下拉框中选择 UTC 选项。再次返回 Discover 面板，时区就调整正确了，日志时间与现在时间相同。

默认的日志包含 `Time` 与 `_source` 两列，`Time` 是日志时间，`_source` 是整个日志数据。我们可根据实际情况，选择感兴趣的日志字段，比如 `message`、`program`。其中，`message` 表示日志内容，`program` 表示日志所对应的程序名称，实际上就是我们在启动 Docker 时指定的日志标签（`--log-opt tag`）。当然，我们也能在 Logstash 中添加更多需要收集的日志字段，比如 `ip`、`container_id` 等。

我们可在 Discover 面板顶部的输入框中输入想要搜索的关键字，将搜索出对应的日志并加亮相应的关键字。我们也能设置 `Time Range` 来搜索指定时间范围内的日志，还能开启 `Refresh Interval` 功能来实现日志的定时刷新。此外，我们也能在 Visualize 面板中自定义一些可视化图表，并将这些图表添加到 Dashboard 面板中。关于 Kibana 的功能还有很多，在实际工作中我们还需不断学习。

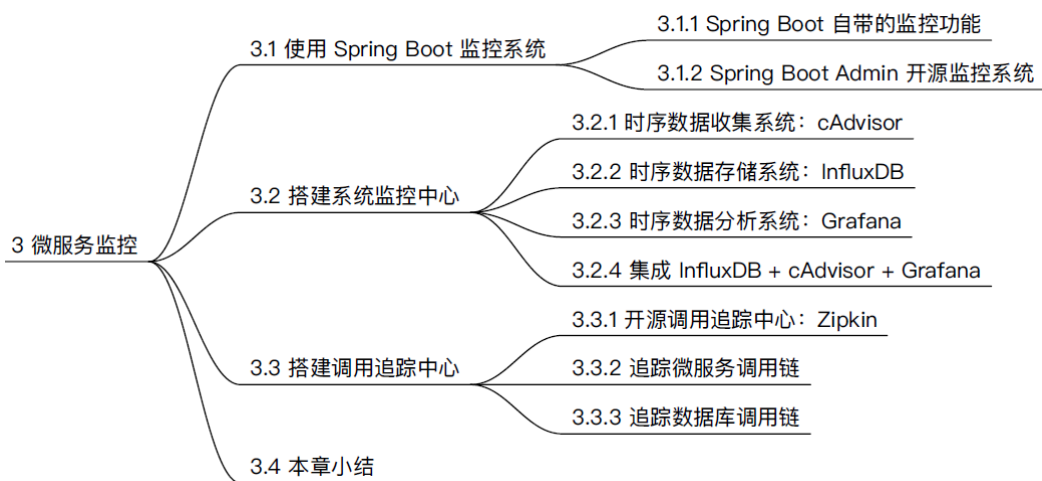
2.4 本章小结

本章的关注点放在微服务日志上，日志是整个微服务应用程序的必备要素。首先我们从 Spring Boot 日志框架入手，使应用日志可以输出到 Docker 容器外部，以便我们可随时查看日志文件，可以帮助我们更加容易地定位应用程序运行过程中所出现的异常现象。随后我们学习了 Docker 日志驱动，并使日志信息输出到 Linux 的 Syslog 中，我们借用了 Syslog 作为应用程序日志的传输通道。最后我们整合了 Syslog 与 ELK 技术栈，成功搭建了一款微服务的“日志中心”，所有微服务运行时所产生的日志都将聚合至 ELK 日志中心。

下一章我们将学习微服务监控方面的技术，目标是能通过图形化界面方式来监控每个微服务的运行状态。

3 chapter

第 3 章 微服务监控



3.1 使用 Spring Boot 监控系统

现在我们已经使用 Spring Boot 开发了许多微服务，并将这些微服务分别封装在不同的 Docker 容器中，最后通过 Jenkins 将微服务自动部署到生产环境中。那么问题来了，如何才能监控生产环境中所运行的微服务呢？我们能够想到的最简单的方式是，通过 `docker stats` 命令在终端上查看每个 Docker 容器的资源使用统计信息，包括 CPU、内存、网络与磁盘使用情况等，但这种方式缺乏图形化界面，我们无法更加直观地看到这些系统的监控信息。因此，我们急需一款基于图形化界面的系统监控系统，它能实时监控 Spring Boot 应用程序的运行状况。通过之前对 Spring Boot 的学习，我们知道有一个叫 Actuator 的插件，可用它来输出 Spring Boot 应用程序中的端点（Endpoint），包括健康检查（/health）、运行指标（/metrics）、线程信息（/dump）等，这些信息正是我们需要的监控数据。

下面我们先快速回顾一下 Spring Boot 自带的 Actuator 插件的基本使用方法。

3.1.1 Spring Boot 自带的监控功能

为了开启 Actuator 插件，我们需要在 Maven 配置文件 `pom.xml` 中添加如下依赖配置。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

当启动 Spring Boot 应用程序后，在浏览器中输入相关端点地址，就能看到相关的系统监控信息。但是，此时只能访问 /health 与 /info 两个端点，其他端点均没有权限访问，因为它们敏感的（Sensitive）。若想访问指定端点，则需在 Spring Boot 的配置文件 `application.properties` 中添加相关配置项。以访问 /metrics 端点为例，我们需要添加如下配置，才能正常访问该端点。

```
endpoints.metrics.sensitive=false
```

若想访问所有端点，则需将所有端点的 sensitive 配置都设为 false，就像下面这样。

```
endpoints.sensitive=false
```

此时，我们便可访问 Actuator 插件提供的所有端点了。不得不说的是，这种做法并不是我们所推荐的最佳实践。一般情况下，我们会尽可能少地对外开放仅有的端点，除非有额外需求，否则能不开放尽量不开放。

我们可通过以下端点，查看 Spring Boot 应用程序当前的运行情况。

- `/info`: 查看应用程序基本信息，需在 `application.properties` 中提供。
- `/health`: 查看应用程序是否健康，即当前是否可以访问。
- `/metrics`: 查看应用程序相关的运行指标，也可自行扩展其他新指标。
- `/env`: 查看应用程序可访问的环境变量，例如 `java.home`、`user.home` 等。
- `/loggers`: 查看或修改应用程序的日志级别配置。
- `/dump`: 查看应用程序的线程相关信息。
- `/trace`: 查看应用程序的请求调用轨迹信息。

以上这些端点可通过浏览器或 HTTP 客户端（例如 Postman）来查看，同样也缺乏图形化的能力。我们很想找到一款软件，它可调用 Spring Boot 应用程序的端点，并自动收集系统运行情况的数据，最后还能以用户界面的方式将这些数据呈现出来，以便我们更方便地观察数据。

幸运的是，我们找到了一款名为 Spring Boot Admin 的开源监控系统，它就能满足我们上面提到的需求。

3.1.2 Spring Boot Admin 开源监控系统

Spring Boot Admin 提供了一个简单的管理界面，它用于管理系统中运行的 Spring Boot 应用程序。它的功能非常丰富，主要包括以下方面。

- 查看健康状态；
- 查看 JVM 与内存指标；
- 查看计数器与计量表指标；
- 查看数据源指标；
- 查看缓存指标；
- 查看系统构建信息；
- 查看并下载日志文件；
- 查看 JVM 系统属性与环境变量；
- 查看线程堆栈信息；
- 查看调用轨迹信息；
- 查看状态变更日志；
- 修改日志级别管理（基于 Logback）；

- 修改 JMX 参数；
- 下载 JVM 堆内存文件；
- 通知状态变更（邮件或即时通信工具）。

如果想进一步学习 Spring Boot Admin，可通过它的 GitHub 站点来获取它的源码。

Spring Boot Admin 源码：<https://github.com/codecentric/spring-boot-admin>。

此外，Spring Boot Admin 的 GitHub 站点上也提供了一份文档，便于我们快速学习它的使用方法。

Spring Boot Admin 文档：<http://codecentric.github.io/spring-boot-admin/1.4.6/>。

从文档中我们可以得知，Spring Boot Admin 实际上包括两部分。第一部分是服务端，它是一个应用程序，用于收集其他 Spring 应用程序的监控信息。第二部分是客户端，需要我们在 Spring Boot 应用程序中进行相关配置，才能在运行时与服务端建立通信，将自身应用程序注册到 Spring Boot Admin 的服务端中。

下面，我们首先来搭建 Spring Boot Admin 服务端。

首先我们需要创建一个 Spring Boot Admin 的 Maven 应用程序，并在 pom.xml 中添加以下依赖配置。

```
<dependency>
  <groupId>de.codecentric</groupId>
  <artifactId>spring-boot-admin-server</artifactId>
  <version>1.4.6</version>
</dependency>

<dependency>
  <groupId>de.codecentric</groupId>
  <artifactId>spring-boot-admin-server-ui</artifactId>
  <version>1.4.6</version>
</dependency>
```

以上有两个 Maven 依赖，spring-boot-admin-server 提供了 Spring Boot Admin 服务端的编程接口，可在代码中直接使用。spring-boot-admin-server-ui 封装了一个 Web 应用程序，该程序使用 Node.js 编写，实际上底层使用了 Maven 来运行 Node.js 应用程序。

完成了 Maven 配置后，接下来需要创建一个 Spring Boot 应用程序启动类，将其命名为

SpringBootAdminApplication，该类的名称可以任意设定。

```
package demo.msa.sba;

import de.codecentric.boot.admin.config.EnableAdminServer;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@EnableAdminServer // 启用 Spring Boot Admin
@SpringBootApplication
public class SpringBootAdminApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringBootAdminApplication.class, args);
    }
}
```

我们需要在该类中添加@EnableAdminServer 注解，在启动该应用程序时，Spring Boot Admin 服务端将扫描该注解，并启动 Spring Boot Admin 应用程序，即运行底层的 Node.js 应用程序。

运行应用程序并打开浏览器，在地址栏中输入 <http://localhost:8080/>，此时可打开 Spring Boot Admin 应用程序的主界面，如图 3-1 所示。

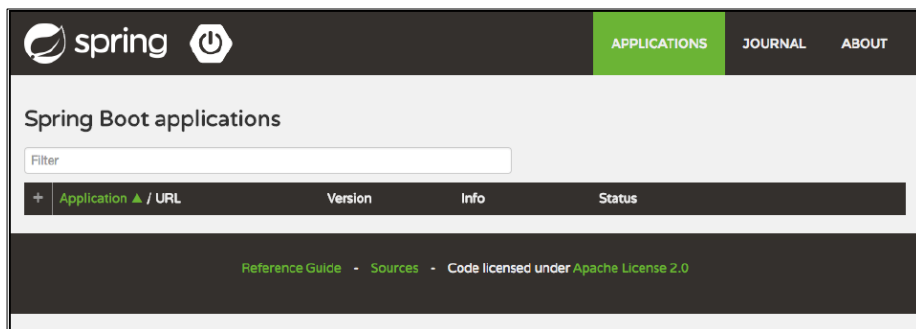


图 3-1 Spring Boot Admin 主界面

此时我们在 APPLICATIONS 界面中看到了一个空的 Spring Boot applications 表格（简称“应用表格”），稍后会在应用表格中加载 Spring Boot Admin 所管理的 Spring Boot 应用程序。

下面我们要做的是创建一个 Spring Boot 应用程序，并使用 Spring Boot Admin 所提供的客户端，在启动 Spring Boot 应用程序时，连接 Spring Boot Admin，并将自己注册到 Spring Boot Admin 中。

我们再创建一个 Maven 应用程序，将其作为一个普通的 Spring Boot 应用程序。首先需要在 pom.xml 中添加 Spring Boot 应用程序所需要的 Maven 依赖，然后再添加以下 Spring Boot Admin 客户端的依赖配置。

```
<dependency>
  <groupId>de.codecentric</groupId>
  <artifactId>spring-boot-admin-starter-client</artifactId>
  <version>1.4.6</version>
</dependency>
```

下面要做的是，在 Spring Boot 应用程序的配置文件 application.properties 中添加 Spring Boot Admin 客户端所需的配置项。

```
spring.boot.admin.url=http://localhost:8080
spring.boot.admin.client.name=HelloService
```

通过 spring.boot.admin.url 配置项来指定 Spring Boot Admin 的连接地址，此外还通过 spring.boot.admin.client.name 指定了该 Spring Boot 应用程序作为 Spring Boot Admin 客户端的名称，若缺少该配置，则 Spring Boot Admin 客户端将从 spring.application.name 配置项中读取。

由于 Spring Boot Admin 占用了本地的 8080 端口，因此我们需要手工指定 Spring Boot 应用程序的端口，不妨将其设置为 8081，继续在 application.properties 中添加如下配置：

```
server.port=8081
```

运行应用程序，随后可在 Spring Boot Admin 的应用列表中看到已注册到 Spring Boot Admin 中的应用程序，如图 3-2 所示。

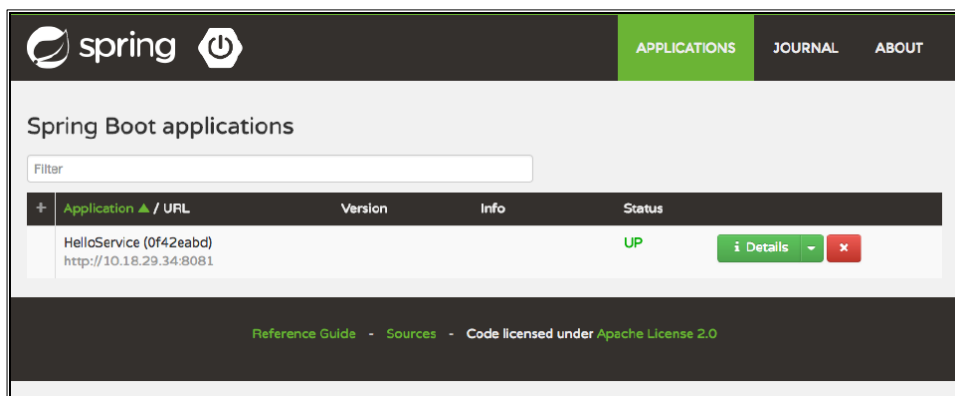


图 3-2 注册 Spring Boot 应用程序

该应用程序名称为 `HelloService`, Spring Boot Admin 还为其生成了一个唯一编号(`0f42eabd`), 还包括该应用程序的访问 URL (`http://10.18.29.34:8001`)。可在应用程序表格中清晰地看到, 当前应用程序的状态为 `UP`, 表示当前可用。我们可点击 `Details` 按钮来查看该应用程序的详细监控信息, 也能点击红色的 `x` 按钮删除该应用程序。

需要说明的是, 应用表格中的 `Version` 与 `Info` 字段均为空白, 实际上我们可从 `Maven` 配置文件中获取该信息, 并注册到 Spring Boot Admin 中, 可以像下面这样配置。

```
info.groupId=@project.groupId@
info.artifactId=@project.artifactId@
info.version=@project.version@
```

这里提供了三个配置项, 每个配置项都以 `info` 作为前缀, 我们从 `pom.xml` 中获取 `project.groupId`、`project.artifactId`、`project.version`, 并将这些参数分别设置到以上三个配置项中。当然, 也能添加其他的 `info` 配置项, 这些配置项均会出现在 Spring Boot Admin 应用表格的 `Info` 字段中。

重启应用程序, 便可在应用表格中看到 `Version` 与 `Info` 字段的数据了, 如图 3-3 所示。

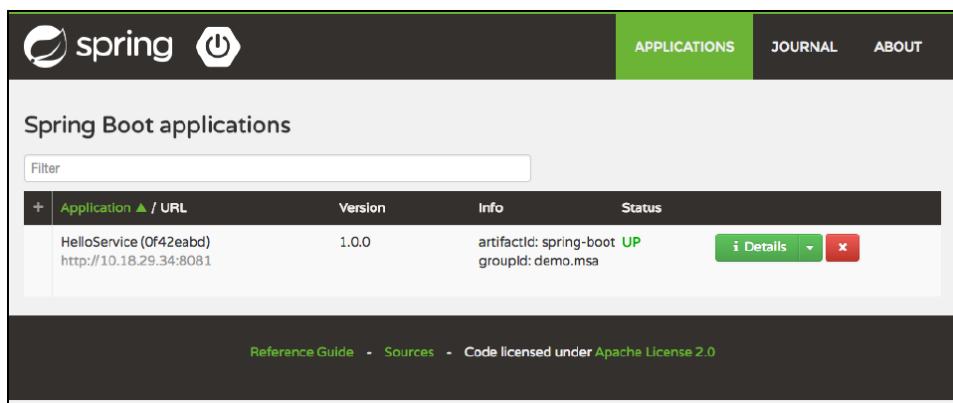


图 3-3 注册 Spring Boot 应用程序

我们点击 `Details` 按钮, 将进入更为详细的监控界面。如果我们没有将相关的端点开放出来, 那么将会看到以下无权限访问详细监控信息的报错信息, 如图 3-4 所示。

```
Error:
{"timestamp":1489558848326,"status":401,"error":"Unauthorized","message":"Full authentication is required to access this resource.","path":"/metrics"}
```

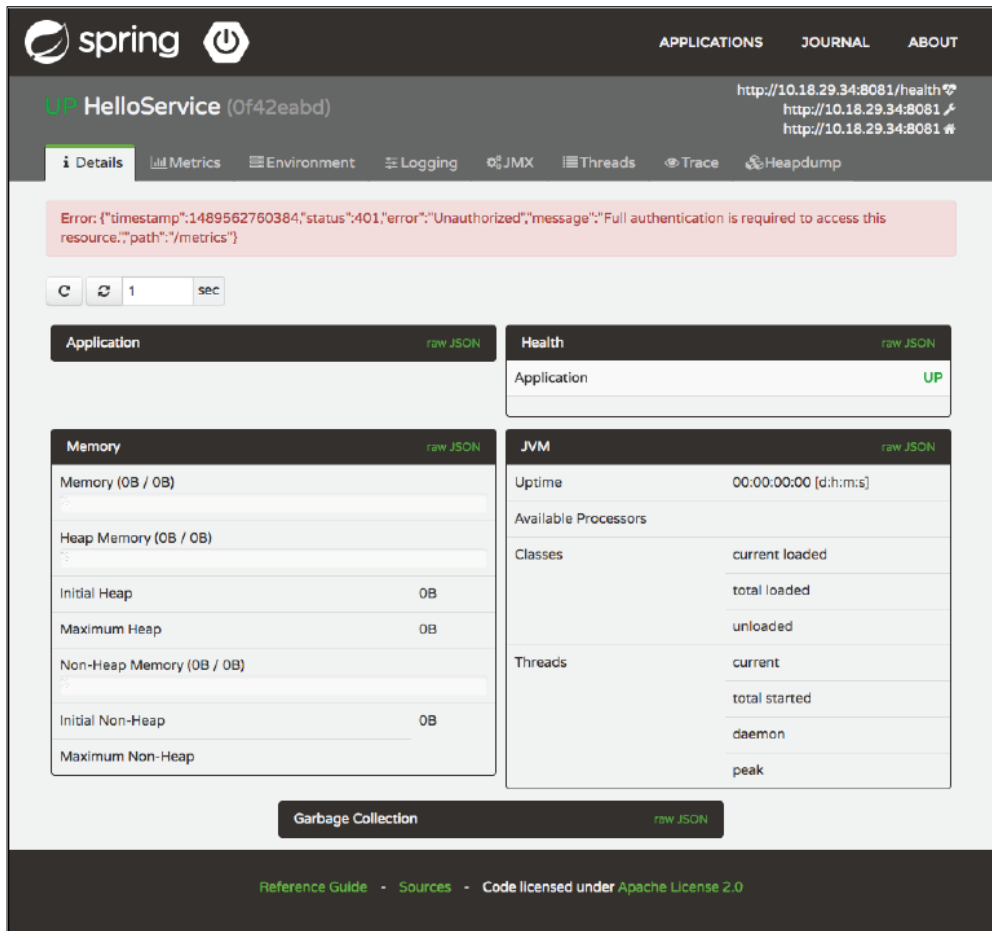


图 3-4 无权限访问详细监控信息

以上报错信息表示当前无权限访问/metrics 端点，我们需要在 application.properties 中手工进行配置。

```
endpoints.metrics.sensitive=false
```

此外，还需要开放其他端点，否则其他选项卡中也会出现报错信息。

```
...
endpoints.env.sensitive=false
endpoints.dump.sensitive=false
endpoints.trace.sensitive=false
```


当然，也可以一次性开启所有的端点。

```
endpoints.sensitive=false
```

重启 Spring Boot Admin 后，将不会再看到类似无权限访问详细监控信息的报错信息，如图 3-5 所示。

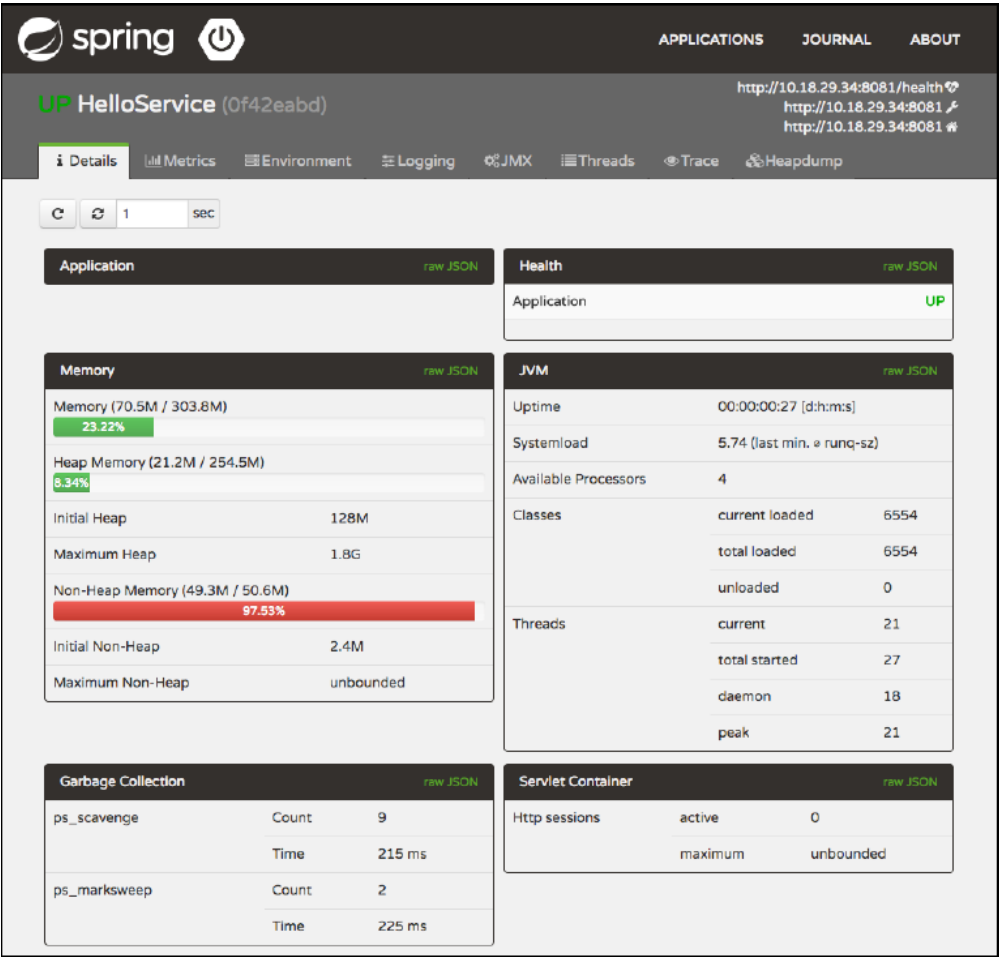


图 3-5 查看详细监控信息

我们可在 Logging 选项卡中动态设置 Spring Boot 应用程序的日志级别，只需在 Spring Boot 项目的 src/main/resources 目录下添加一个 Logback 的配置文件 logback.xml，就能开启日志监控功能。

```
<configuration>
  <include resource="org/springframework/boot/logging/logback/base.xml"/>
  <jmxConfigurator/>
</configuration>
```

重启 Spring Boot 应用程序后，将在 Logging 选项卡中看到一个日志级别的管理控制台，包含 Spring Boot 应用程序所有包名对应的日志级别，此时可修改相应的日志级别，如图 3-6 所示。

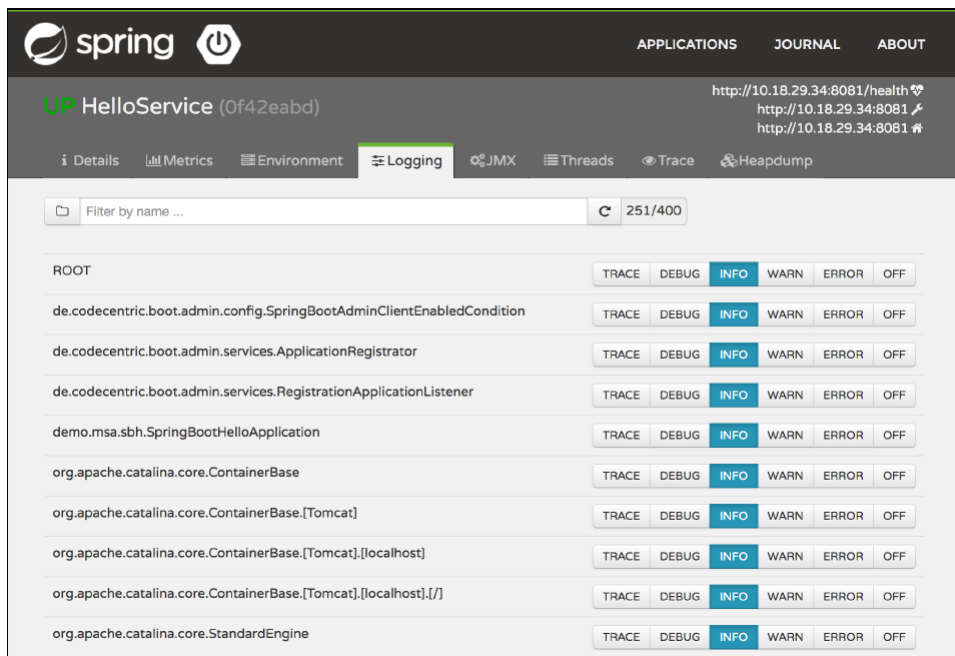


图 3-6 日志级别管理控制台

点击 JMX 选项卡，此时我们将看不到任何 JMX 监控数据，只需在 Spring Boot 应用程序中添加如下 Maven 依赖配置，就能开启 JMX 监控功能。

```
<dependency>
  <groupId>org.jolokia</groupId>
  <artifactId>jolokia-core</artifactId>
</dependency>
```

重启 Spring Boot 应用程序后，将在 JMX 选项卡中看到一个 JMX 管理控制台，包含 Spring Boot 应用程序所暴露的所有 JMX 参数，此时可修改相应的 JMX 参数，如图 3-7 所示。

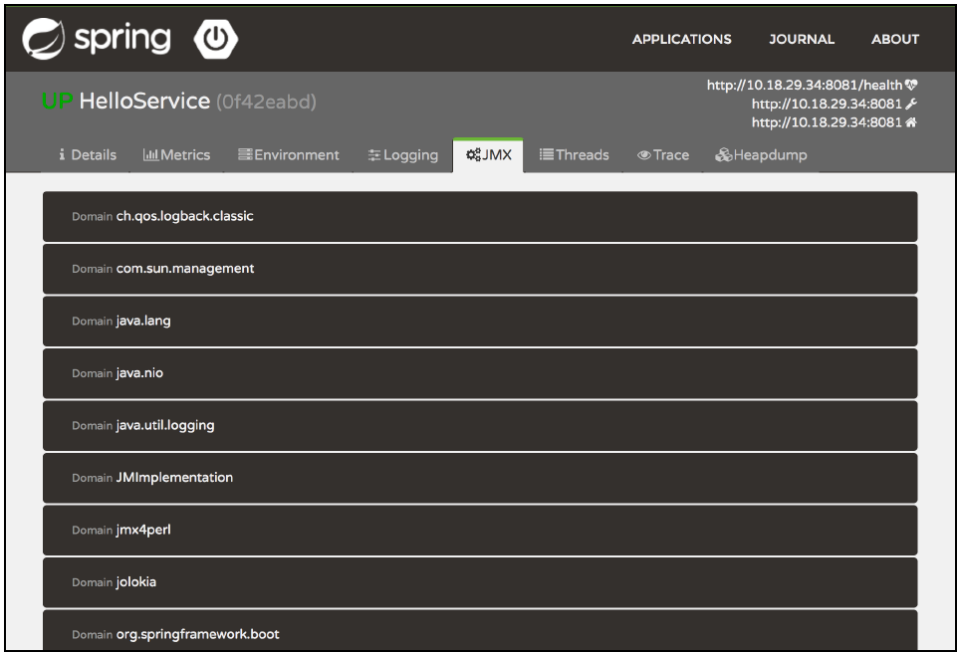


图 3-7 JMX 管理控制台

打开 JOURNAL 界面，在该界面中会看到一个 Journal 表格（简称“日志表格”），该表格中会显示 Spring Boot 应用程序的事件变化，如图 3-8 所示。

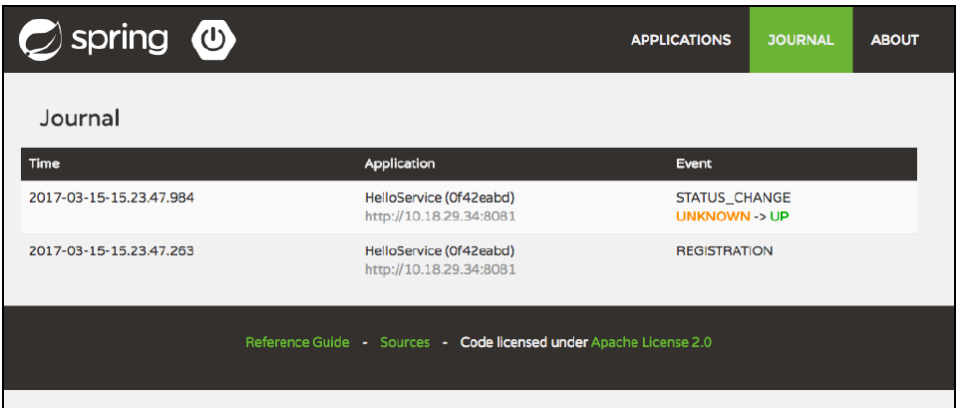


图 3-8 Spring Boot 应用日志

当前的日志表格中有两条数据，根据 Time 字段倒序排列。最下面一条记录表示应用程序刚注册，此时产生了一个注册事件(REGISTRATION)；第一条记录表示状态由未知(UNKNOWN)变为上线 (UP)，此时产生了一个状态变更事件 (STATUS_CHANGE)。当 Spring Boot 应用程

序停止后，同样也会产生一个状态变更事件，此时的状态从 UP 变为 OFFLINE，重启后状态又从 OFFLINE 变为 UP。总之，Spring Boot Admin 会监控并记录 Spring Boot 应用程序所有的运行状态。

目前的 Spring Boot Admin 是没有任何安全控制的，为了防止非法访问，我们可开启 Spring Security 的用户身份认证功能。需要分别在 Spring Boot Admin 的服务端与客户端中配置相应的用户名与密码（可根据实际情况自行设置），首先需要在服务端中的 application.properties 中添加如下配置。

```
security.user.name=admin  
security.user.password=admin
```

此外，我们还需要在 pom.xml 中添加 Spring Security 的 Maven 依赖配置。

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-security</artifactId>  
</dependency>
```

随后，我们还需要在 Spring Boot Admin 的客户端应用的 application.properties 中添加对应的用户名与密码（需要与服务端保持相同设置），因为 Spring Boot 应用程序在启动时需要通过 Spring Boot Admin 的用户身份认证。

```
spring.boot.admin.username=admin  
spring.boot.admin.password=admin
```

重启 Spring Boot Admin 与相应的客户端应用，重新打开浏览器并再次访问 Spring Boot Admin 时，此时将弹出一个对话框，我们需要输入正确的用户名与密码，才能通过用户身份认证。

关于 Spring Boot Admin 的功能点还有很多，先介绍到这里，大家有兴趣可以从它的官网文档上了解更多内容。

既然应用程序可以被监控，那么应用程序所在的操作系统也应该需要监控，也就是说，我们需要对微服务所在的 Docker 容器也进行监控，这样才能更加全面地了解整个系统的运行状况。下面我们一起来搭建一款系统监控中心，它将负责 Docker 容器的系统监控。

3.2 搭建系统监控中心

之前我们讨论了关于微服务内部运行状况的监控,但还未涉及微服务所在操作系统的监控,我们下面要做的就是搭建一个微服务系统监控中心,该平台会不断收集每个微服务所在 Docker 容器中随时间变化的数据(简称“时序数据”),包括 CPU、内存、网络、磁盘等使用情况。我们同样也需要使用一系列开源技术来搭建这款系统监控中心,该平台首先会从 Docker 容器中收集相关时序数据,随后会将这些时序数据存入一个时序数据库中,最后通过一个 Web 应用程序以图表的方式来展示并分析这些时序数据。可见,系统监控中心包括了三个系统,即时序数据收集系统、时序数据存储系统、时序数据分析系统,这三个系统需要天然无缝地整合在一起。经过一番调研后决定使用 cAdvisor 实现时序数据收集,使用 InfluxDB 实现时序数据存储系统,使用 Grafana 实现时序数据分析。

下面,我们先来学习 cAdvisor 时序数据收集系统。

3.2.1 时序数据收集系统: cAdvisor

cAdvisor 是 Google 推出的一款基于 Go 语言的开源产品,它用于分析 Docker 容器在运行中的资源使用与性能特征。cAdvisor 使用起来非常简单,我们可以通过启动 Docker 容器的方式来运行它,当它运行后即可监控当前宿主机中所有 Docker 容器的运行状况。

可从 GitHub 站点上获取 cAdvisor 的源码,还能快速了解它的相关特性。

cAdvisor 源码: <https://github.com/google/cadvisor>。

我们可使用以下 `docker run` 命令来启动 cAdvisor 容器,启动时它将保持在后台运行。

```
docker run \
-d \
-p 8080:8080 \
-v /:/rootfs \
-v /var/run:/var/run \
-v /sys:/sys \
-v /var/lib/docker:/var/lib/docker \
--name=cadvisor \
google/cadvisor
```

cAdvisor 提供了一个 Web 控制台,以便于我们通过图形化的方式来查看它所收集的时序数据,我们可在浏览器上通过 8080 端口来访问该控制台。在启动 cAdvisor 容器时,我们需要同

时挂载以上 4 个本地目录到容器中。需要注意的是，如果使用 CentOS 操作系统作为宿主机，还需挂载/cgroup 目录并添加--privileged 选项，否则 cAdvisor 容器无法正常启动。

```
docker run \  
-d \  
-p 8080:8080 \  
-v /:/rootfs \  
-v /var/run:/var/run \  
-v /sys:/sys \  
-v /var/lib/docker:/var/lib/docker \  
-v /cgroup:/cgroup \  
--privileged \  
--name=cadvisor \  
google/cadvisor
```

cAdvisor 容器启动后，我们可在浏览器上访问 <http://localhost:8080/>，来查看 cAdvisor 自带的 Web 控制台，如图 3-9 所示。

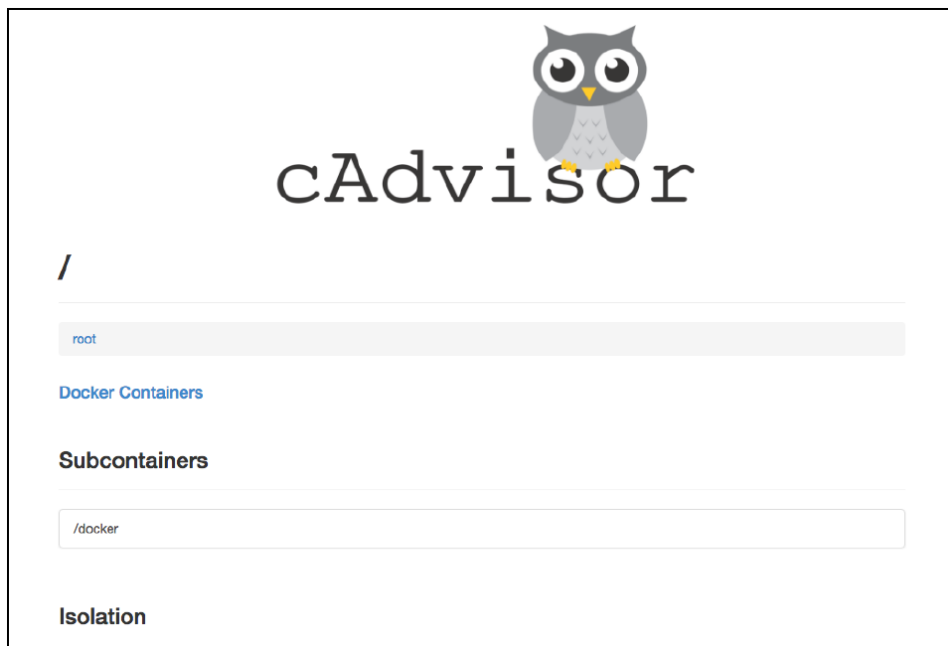


图 3-9 cAdvisor 自带的 Web 控制台

我们可通过 cAdvisor 所提供的 Web 控制台观察宿主机 CPU 的整体使用情况，还能观察每

个 CPU 内核的使用情况，如图 3-10 所示。

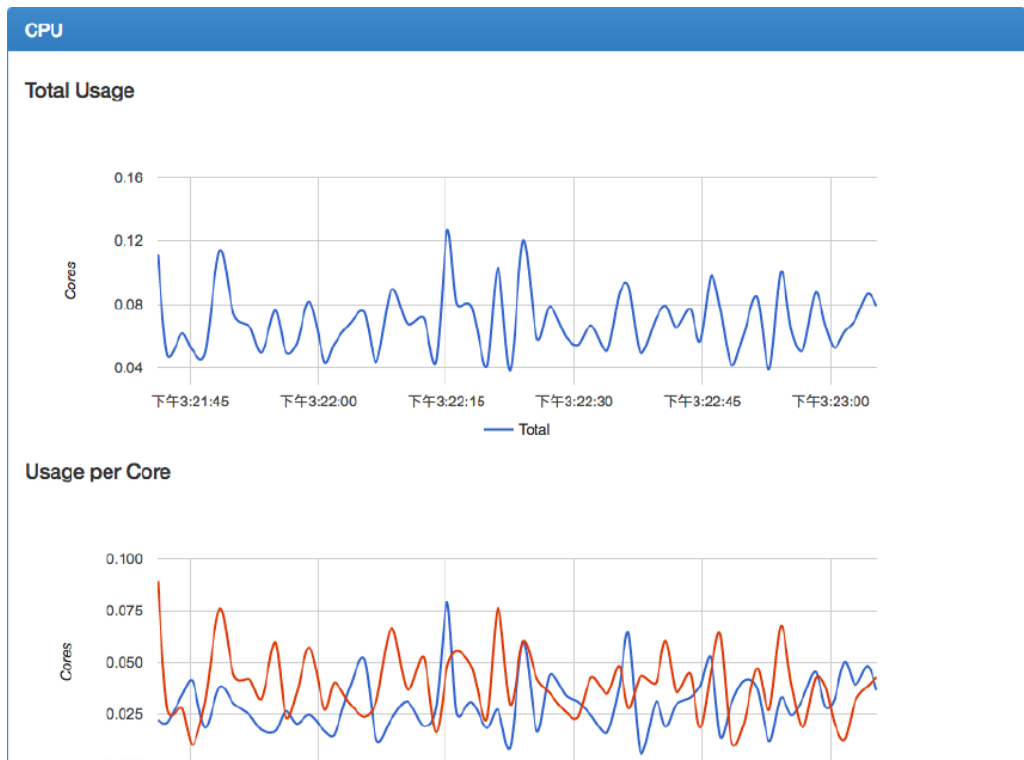


图 3-10 cAdvisor 监控 CPU 使用情况

我们不仅可以监控当前宿主机的运行状况，也能针对指定的 Docker 容器来查看每个容器的运行状况，当然也能监控 cAdvisor 容器自身的运行状况。

除此之外，cAdvisor 还提供了一套 REST API，我们可通过浏览器、Postman 客户端、应用程序来访问它的 REST API。我们可利用 cAdvisor 所提供的 REST API 自行封装客户端 API，这样更有利于程序开发，官方也提供了一个基于 Go 语言的客户端 API。

cAdvisor REST API: <https://github.com/google/cadvisor/blob/master/docs/api.md>。

cAdvisor 客户端 API: <https://github.com/google/cadvisor/blob/master/docs/clients.md>。

cAdvisor 虽然使用起来非常方便，通过它自带的 Web 控制台也能基本满足我们的监控需求，但仍然存在以下两个问题。

（1）只能监控当前宿主机的运行状况，如何监控远程主机呢？

（2）只能看到当前一段时间内的时序数据，如何长期存储呢？

我们可在多台宿主机上分别启动 cAdvisor 容器，这样可解决第一个问题，但仍然无法解决第二个问题，因为 cAdvisor 所展现的仅为最近一段时间内的时序数据，无法将所有时序数据长期存储在磁盘中。对于第二个问题，我们非常有必要使用一个数据库来存放每个容器所产生的时序数据，这个存放时序数据库的系统就是 InfluxDB。

3.2.2 时序数据存储系统：InfluxDB

InfluxDB 是 InfluxData 公司的产品之一，它也是一款开源产品，用于存储相应的时序数据。由于时序数据是随时间变化而产生的，每分每秒（甚至每毫秒）都会产生相应的数据，而且这个数据量还会不断积累，最终将形成一个庞大的数据集。InfluxDB 天生就用来存储这些大数据，同时还能对外提供高效的数据查询功能。

可从 InfluxData 公司官网上了解 InfluxDB 的相关介绍，也能获知 InfluxData 公司的更多产品与服务，如图 3-11 所示。

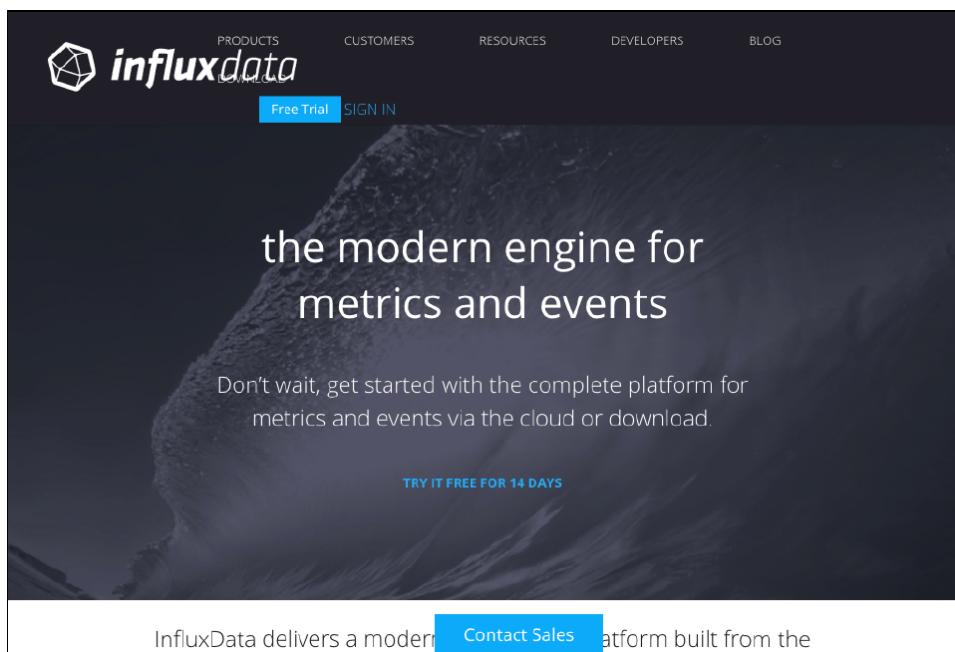


图 3-11 InfluxData 官网

InfluxData 公司: <https://www.influxdata.com/>。

可从 GitHub 站点上获取 InfluxDB 的源码，还能快速了解它的相关特性。

InfluxDB 源码: <https://github.com/influxdata/influxdb>。

使用 InfluxDB 有两个步骤，首先要做的是启动 InfluxDB 服务端，随后可通过它所提供的 REST API 或命令行客户端来访问 InfluxDB 服务端。

当然，我们依旧使用 `docker run` 的方式启动 InfluxDB 容器。

```
docker run \
-d \
-p 8086:8086 \
-v ~/influxdb:/var/lib/influxdb \
--name influxdb \
influxdb
```

此时对外暴露的 8086 端口就是 InfluxDB 所提供的 REST API 端口，此外还需将容器内部的数据目录（`/var/lib/influxdb`）映射到宿主机目录上（`~/influxdb`），这样有助于数据备份与还原。

当 InfluxDB 容器启动后，我们便可使用 `docker exec` 命令进入 InfluxDB 容器内部，并执行 `influx` 命令，来启动 InfluxDB 的命令行客户端。

```
docker exec -it influxdb influx
```

此时将看到一个 InfluxDB 的版本以及命令行提示符，我们随后将输入相关 InfluxDB 命令来操作 InfluxDB 数据库。

```
InfluxDB shell version: 1.2.2
>
```

输入 `exit` 可退出 InfluxDB 命令行客户端，输入 `help` 可查看相关使用帮助。

```
connect <host:port>    connects to another node specified by host:port
auth                  prompts for username and password
pretty                toggles pretty print for the json format
use <db_name>         sets current database
format <format>        specifies the format of the server responses: json,
csv, or column
precision <format>    specifies the format of the timestamp: rfc3339, h, m,
s, ms, u or ns
```

```
consistency <level>  sets write consistency level: any, one, quorum, or all
history              displays command history
settings             outputs the current settings for the shell
clear                clears settings such as database or retention
policy. run 'clear' for help
exit/quit/ctrl+d     quits the influx shell
show databases        show database names
show series           show series information
show measurements     show measurement information
show tag keys         show tag key information
show field keys       show field key information
```

首先，我们输入 `SHOW DATABASES` 命令，看看 InfluxDB 中有哪些数据库。

```
> SHOW DATABASES
name: databases
name
----
_internal
```

此时，InfluxDB 中只有一个名为 `_internal` 的数据库，它是 InfluxDB 内部的数据库，一般情况下我们不会使用它。

现在，我们使用 `CREATE DATABASE` 命令来创建一个新的数据库。

```
> CREATE DATABASE mydb
```

此时没有任何输出消息，表明“没消息就是好消息”。

接着，我们继续使用 `SHOW DATABASES` 命令来查看新创建的数据库是否已经在 InfluxDB 中。

```
> SHOW DATABASES
name: databases
name
----
_internal
mydb
```

此时看到了我们刚刚创建的 `mydb` 数据库，说明之前的创建数据库操作成功了。

接着，使用 USE 命令来选中当前新创建的数据库。

```
> USE mydb
Using database mydb
```

通过输出的消息，我们便知操作成功了。

接着，我们使用 INSERT 命令插入一条时序数据到当前选中的数据库中。

```
> INSERT cpu,host=serverA,region=us_west value=0.64
```

以上 INSERT 命令所插入的数据中，实际上包含三部分，即 measurement、tag、field。

(1) measurement：表示量度，类似关系型数据库中的数据表（如 cpu）。

(2) tag：表示标签，类似关系型数据库中的索引，tag 用 key=value 结构表示，每个 tag 用 “,” 分隔（如 host=serverA,region=us_west）。

(3) field：表示字段，类似关系型数据库中的字段，field 用 key=value 结构表示（如 value=0.64）。

需要注意的是，每条时序数据中，量度只能有一个，但标签与字段可以有多个。

接着，我们使用 SELECT 命令来查询之前刚插入的一条时序数据。

```
> SELECT "host", "region", "value" FROM "cpu"
name: cpu
time                host      region  value
----                -
1489655425837548060 serverA us_west 0.64
```

我们在 cpu 量度中查询出 host 与 region 标签以及 value 字段，其中第一列 time 表示该量度的创建时间，它由 InfluxDB 自动生成。当然，此时的 time 是一个长整型数字，实际上它是一个相对毫秒数，我们无法用肉眼所识别。如果想用可读性更好的方式来查看 time 列，那么需要在进入 InfluxDB 命令行客户端时带上 -precision rfc3339 参数，表示时间精度使用 RFC3339 规范来输出。

```
docker exec -it influxdb influx -precision rfc3339
```

我们退出 InfluxDB 命令行客户端后，再次使用 SELECT 命令就能查看 RFC3339 格式的 time 列了。

```
SELECT "host", "region", "value" FROM "cpu"
```

```
name: cpu
time                host      region  value
----                -
2017-03-16T09:10:25.83754806Z serverA us_west 0.64
```

下面，我们再使用 INSERT 命令插入一条时序数据。

```
> INSERT temperature,machine=unit42,type=assembly external=25,internal=37
```

此时插入的时序数据中，量度为 temperature，标签为 machine=unit42,type=assembly，字段为 external=25,internal=37。

接着，通过 SELECT * 可查看 temperature 量度的所有标签与字段。

```
> SELECT * FROM "temperature"
name: temperature
time                external internal machine type
----                -
2017-03-16T09:20:04.246104923Z 25          37          unit42  assembly
```

以上 SELECT 命令只能针对某个量度来查询，如何一次性查询所有量度呢？实际上，我们可在 InfluxDB 中使用正则表达式来表示需要查询的量度名称，也能使用 LIMIT 关键字指定需要输出的时序数据条数。

```
> SELECT * FROM /.*/ LIMIT 1
name: cpu
time                external host  internal  machine region  type value
----                -
2017-03-16T09:10:25.83754806Z          serverA          us_west      0.64

name: temperature
time                external host  internal  machine region type  value
----                -
2017-03-16T09:20:04.246104923Z 25          37          unit42      assembly
```

关于 InfluxDB 命令行客户端更多的用法，请参见它的官方文档。

InfluxDB 命令行客户端: <https://docs.influxdata.com/influxdb/v1.2/tools/shell/>。

此外, InfluxDB 也提供了便于使用的 REST API。

REST API: <https://docs.influxdata.com/influxdb/v1.2/tools/api/>。

我们也可通过封装 InfluxDB 的 REST API 来封装出更利于程序开发的客户端 API, 当然, InfluxDB 官方也为我们提供了许多现成的客户端 API, 有 Go 语言的, 也有 Java 语言的。

客户端 API: https://docs.influxdata.com/influxdb/v1.2/tools/api_client_libraries/。

InfluxDB 的基本用法就先介绍到这里, 关于更多高级的用法, 大家可以从 InfluxDB 的官方文档中学习。

InfluxDB 官方文档: <https://docs.influxdata.com/influxdb/v1.2/>。

需要补充说明的是, 虽然 InfluxDB 具备存储超大规模时序数据的能力, 但是它的开源版本却没有提供集群特性。也就是说, 如果大家希望 InfluxDB 具备高可用特性与水平扩展能力, 那么只能使用它的企业版本 InfluxEnterprise。不过在选择使用 InfluxEnterprise 之前, 必须提醒的是, 请先确保自己的时序数据需要长期存储下来, 并真的足够大, 以及无法忍受任何的不稳定性。

当时序数据存储下来以后, 我们要关心的是如何将这些时序数据更直观地展现出来, 从而更进一步对这些时序数据进行分析。Grafana 就是我们进行时序数据分析的利器。

3.2.3 时序数据分析系统: Grafana

Grafana 是一款基于 Web 的开源时序数据分析软件, 它的功能十分强大, 界面非常专业且相当易用。它能无缝对接 InfluxDB 数据源, 非常轻松愉快地就能将我们带入数据可视化的世界。

可从 Grafana 官网上了解它的更多信息, 以及相关使用方法, 如图 3-12 所示。

Grafana 官网: <https://grafana.com/>。

可从 GitHub 站点上获取 Grafana 的源码, 还能快速了解它的相关特性。

Grafana 源码: <https://github.com/grafana/grafana>。



图 3-12 Grafana 官网

我们仍然可以使用 `docker run` 来启动 Grafana 容器，此时需要传入一些 Grafana 所需的环境变量。

```
docker run \
-d \
-p 3000:3000 \
-v ~/grafana:/var/lib/grafana \
--name grafana \
grafana/grafana
```

我们对外暴露了一个 3000 端口，它是 Grafana 的 Web 应用程序端口。此外，我们也需要将 Grafana 的数据目录（`/var/lib/grafana`）映射到宿主机的目录上（`~/grafana`）。

Grafana 容器启动后，在浏览器中访问 `http://127.0.0.1:3000/`，就能看到 Grafana 的登录界面，此时需要输入 Grafana 的管理员用户名和密码，默认的管理用户名是 `admin`，该用户的密码默认也是 `admin`，可在启动 Grafana 容器时，通过指定 `GF_SECURITY_ADMIN_PASSWORD` 环境变量来设置 `admin` 用户的默认密码。

登录 Grafana 后，我们将进入 Home Dashboard 界面，如图 3-13 所示。

在向导中可以看到，当前 Grafana 已经安装完毕，接下来需要添加数据源、创建仪表盘、邀请团队加入、安装应用与插件。

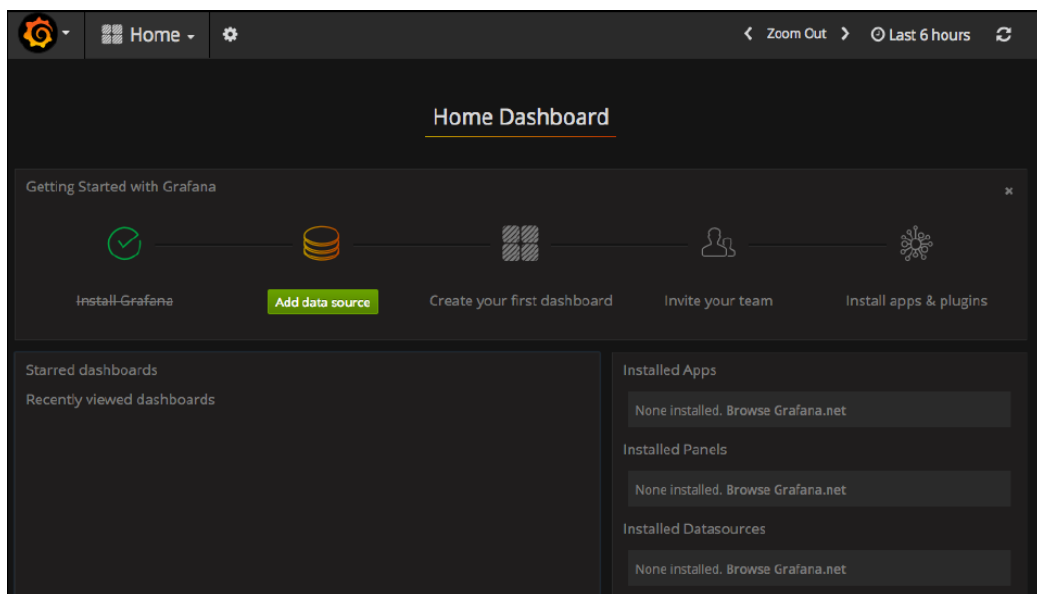


图 3-13 Grafana 主界面

至此，Grafana 可正常使用，下面我们要做的是，将 InfluxDB、cAdvisor 与 Grafana 集成在一起，构建一个系统监控中心。

3.2.4 集成 InfluxDB + cAdvisor + Grafana

我们将以上讲到的 InfluxDB、cAdvisor、Grafana 集成在一起，它们三个系统的组合将构成一个完整的系统监控该平台。InfluxDB 是存放时序数据的中心，cAdvisor 将时序数据存入 InfluxDB，Grafana 从 InfluxDB 中获取时序数据，也就是说，InfluxDB 是整个系统监控中心的数据库，时序数据从 cAdvisor 进行输入，从 Grafana 产生输出。那么我们需要先启动 InfluxDB，随后才能启动 cAdvisor 与 Grafana。

1) 启动 InfluxDB 容器

我们通过以下 docker run 命令来启动 InfluxDB 容器。

```
docker run \
-d \
-p 8086:8086 \
-v ~/influxdb:/var/lib/influxdb \
--name influxdb \
influxdb
```

必须对外暴露一个端口，作为 InfluxDB 的 REST API 端口，默认的端口是 8086。此外，建议将 InfluxDB 的数据文件映射到宿主机中。

启动 InfluxDB 容器成功后，我们需进入该容器内部，并打开 InfluxDB 的命令行客户端。

```
docker exec -it influxdb influx
```

首先需要创建一个名为 `cadvisor` 的数据库，用于存放 cAdvisor 所收集的时序数据。

```
CREATE DATABASE "cadvisor"
```

随后还需创建一个 `root` 用户，密码也为 `root`，该用户拥有所有权限。

```
CREATE USER "root" WITH PASSWORD 'root' WITH ALL PRIVILEGES
```

后续在启动 cAdvisor 容器与配置 Grafana 数据源时，需手工设置 InfluxDB 的数据库及其用户。

2) 启动 cAdvisor 容器

我们通过以下 `docker run` 命令来启动 cAdvisor 容器。

```
docker run \
-d \
-v /:/rootfs \
-v /var/run:/var/run \
-v /sys:/sys \
-v /var/lib/docker:/var/lib/docker \
--link=influxdb:influxdb \
--name=cadvisor \
google/cadvisor \
  -storage_driver=influxdb \
  -storage_driver_host=influxdb:8086 \
  -storage_driver_db=cadvisor \
  -storage_driver_user=root \
  -storage_driver_password=root
```

启动 cAdvisor 容器时，需连接 InfluxDB 容器，并传入 cAdvisor 所需的初始化参数。

- `-storage_driver`: cAdvisor 的存储驱动，需设置为 `influxdb`，表示 InfluxDB。
- `-storage_driver_host`: 设置 InfluxDB 的连接方式，域名:端口。

- -storage_driver_db: 设置 InfluxDB 的数据库。
- -storage_driver_user: 设置 InfluxDB 的用户。
- -storage_driver_password: 设置 InfluxDB 的用户密码。

3) 启动 Grafana

我们通过以下 docker run 命令来启动 Grafana 容器。

```
docker run \  
-d \  
-p 3000:3000 \  
-v ~/grafana:/var/lib/grafana \  
--link=influxdb:influxdb \  
--name grafana \  
grafana/grafana
```

启动 Grafana 容器成功后，接下来需要为 Grafana 添加一个数据源，点击 Home Dashboard 界面中的 Add data source 按钮，随后需手工添加一个 Grafana 数据源，如图 3-14 所示。

The screenshot shows the 'Add data source' configuration page in Grafana. The page has a dark theme. At the top, there's a navigation bar with a gear icon and a 'Data Sources' tab. Below the navigation bar, the title 'Add data source' is displayed. There are two tabs: 'Config' (selected) and 'Dashboards'. The 'Config' tab contains several sections: 'Name' with a text input 'My data source name' and a 'Default' checkbox; 'Type' with a dropdown menu set to 'Graphite'; 'Http settings' with 'Url' set to 'http://localhost:8080' and 'Access' set to 'proxy'; and 'Http Auth' with 'Basic Auth' and 'TLS Client Auth' sections, each containing checkboxes for 'With Credentials' and 'With CA Cert'. At the bottom, there are 'Add' and 'Cancel' buttons.

图 3-14 添加 Grafana 数据源

所有的数据源配置均在 Config 选项卡中完成,在 Name 输入框中填写数据源名称(cadvisor),勾选 Default 复选框(表示 cadvisor 为默认数据源),在 Type 下拉框中选择数据源类型(InfluxDB),在 Url 输入框中填写 InfluxDB 的访问地址 (http://influxdb:8086),在 Access 下拉框中选择连接方式(proxy)。最后在 Database 输入框中填入 cadvisor,在 User 输入框中填入 root,在 Password 输入框中填入 root。点击 Add 按钮, Grafana 的 InfluxDB 数据源就创建完毕了。

下面,我们回到 Home Dashboard 界面,点击 New dashboard 按钮,开始创建一个新的仪表盘。

首先需要选择一种仪表盘类型,我们选择 Graph (图表),随后将看到一张只有横纵坐标的空面板,此时需点击面板标题(Panel Title),将出现一个悬浮菜单,继续点击悬浮菜单中的 Edit 按钮,将在面板的下方看到一个面板编辑表单,如图 3-15 所示。

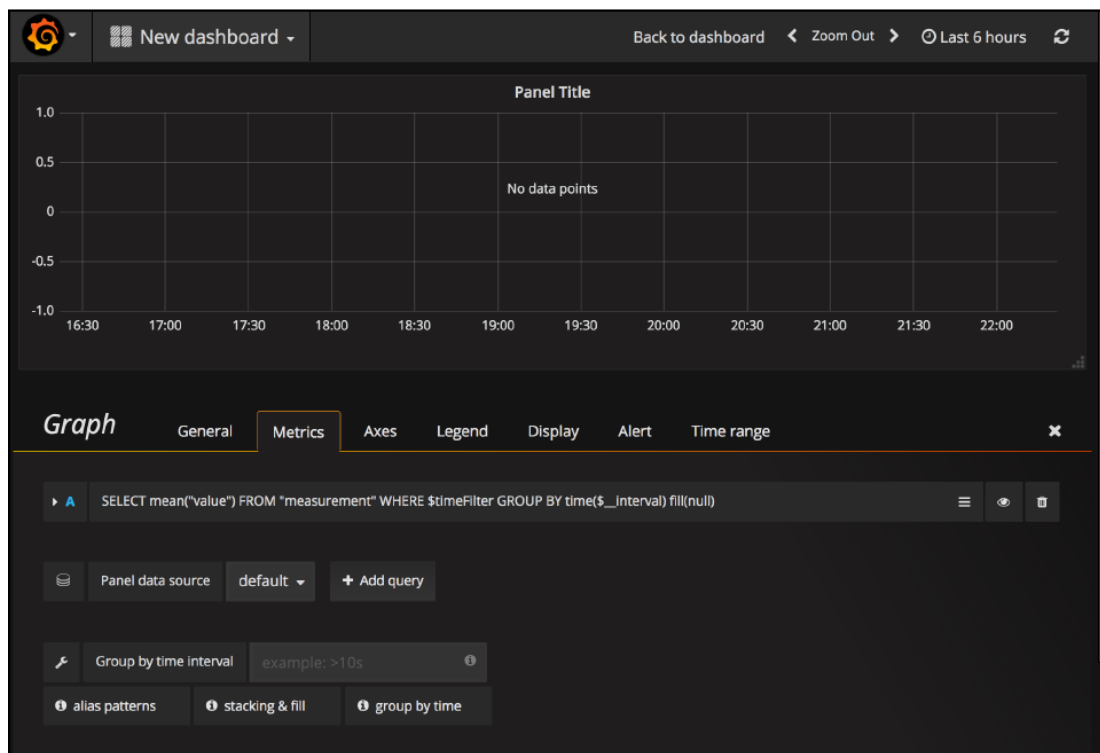


图 3-15 编辑仪表盘面板

在 Metrics 选项卡中有一段 InfluxDB 查询语句,我们点击该查询语句,将出现一个 SELECT 语句的表单,如图 3-16 所示。

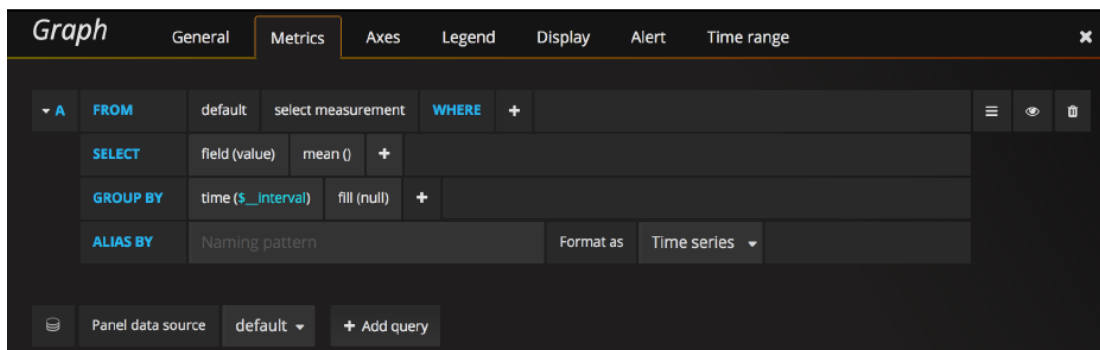


图 3-16 编辑 InfluxDB 查询语句

点击 select measurement 文字，将出现 InfluxDB 默认数据源（cadvisor）的所有量度。

- cpu_usage_per_cpu;
- cpu_usage_system;
- cpu_usage_total;
- cpu_usage_user;
- fs_limit;
- fs_usage;
- load_average;
- memory_usage;
- memory_working_set;
- rx_bytes;
- rx_errors;
- tx_bytes;
- tx_errors。

我们可以选择 memory_usage 来查看内存的使用情况，此时将立即看到面板中有曲线输出。如果仅仅只想查看 Grafana 容器自身的内存使用情况，那么需要在 WHERE 后添加一个“container_name = grafana”的条件。我们可以自行调整时间聚合与时间范围，将动态调整输出的曲线。可以在 General 选项卡中设置面板的标题（Memory Usage），这样看起来更加有意义。还可以在 Axes 选项卡中设置恰当的纵轴单位（Left Y Unit），这里应该设置为 bytes，才能表示内存所占用的字节数。当然，也能在同一面板中添加多条曲线，比如：同时将 InfluxDB、cAdvisor、Grafana 的内存使用曲线放入同一面板中，来对比内存使用情况，如图 3-17 所示。

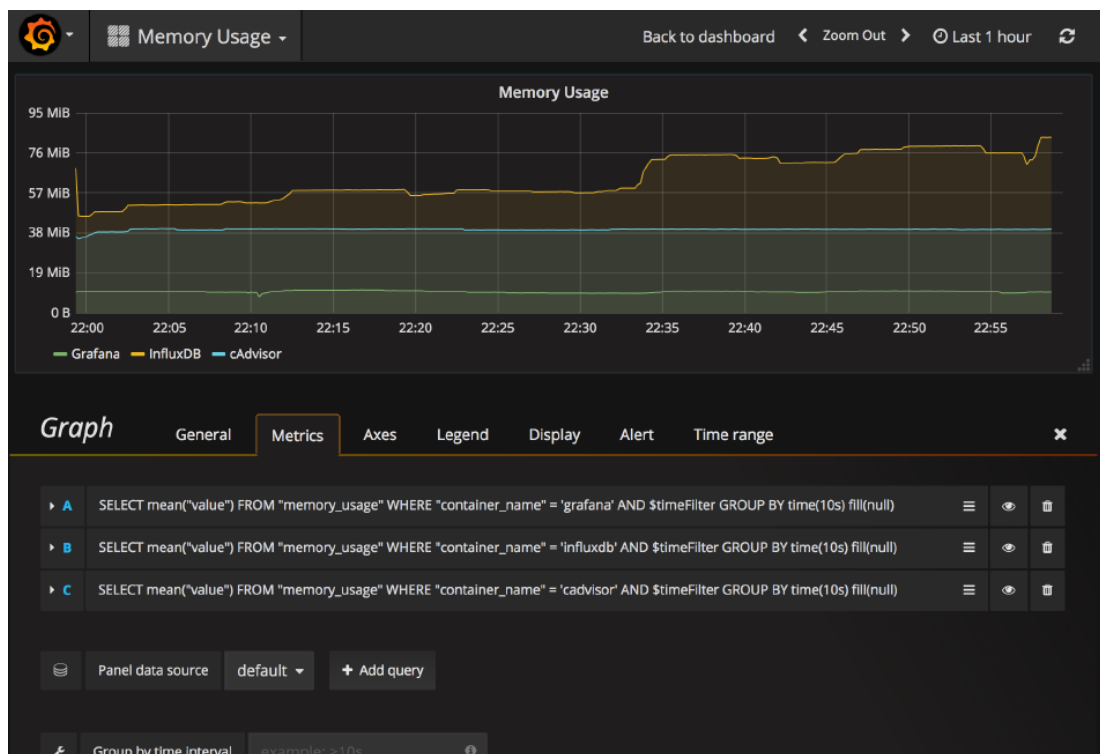


图 3-17 对比内存使用情况

Grafana 的功能非常强大，目前只是制作了一个简单的图表，此外我们还能在 Alert 选项卡中启用报警功能，这个就留给大家自行探索吧。

3.3 搭建调用追踪中心

从微服务架构来看客户端请求的调用轨迹，当客户端发送一个请求后，该请求首先会进入微服务网关，随后通过微服务网关的服务发现机制进行反向代理，从而调用当前可用的微服务。在调用微服务的过程中，简单情况下客户端一次请求只会调用一个微服务 API，但复杂情况下一次请求可能会产生大量的调用，包括微服务与数据库之间的调用，还会有微服务之间的调用。如果我们把每次请求看成一个调用链的话，那么这个调用链上的每个节点都对应一个独立的组件，组件可能是微服务网、具体的微服务、数据库，或者其他中间件等。

那么问题来了，我们应该如何监控整个微服务架构的调用链呢？也就是说，我们想知道一次客户端请求所经历的过程分别调用了哪些组件，哪些微服务。我们还想了解每次客户端请求的总时长，以及调用每个组件分别所花费的时长，通过观察这些时间片段，可以清晰定位到每

次调用过程中的性能瓶颈，有助于我们后续进行性能调优。因此，我们非常有必要在微服务架构中引入一个调用追踪中心，开源的解决方案 Zipkin 应该可以给我们带来帮助。

下面，我们就一起进入 Zipkin 的世界。

3.3.1 开源调用追踪中心：Zipkin

Zipkin 是 Twitter 公司开源的一款调用追踪中心（也称为分布式追踪系统），它可以帮助我们收集分布式系统中每个组件所花费的调用时长，并通过图形化界面的方式来展现整个调用链依赖关系，还能展现调用每个组件所花费的时长。Zipkin 在系统设计上参考了 Google Dapper，它是 Google 公司内部所使用的大规模分布式系统追踪基础设施。

Google Dapper: <https://research.google.com/pubs/pub36356.html>。

可以毫不夸张地说，Zipkin 是开源社区中调用追踪中心的首选方案，我们可从 Zipkin 的官网上了解更多关于它的相关介绍与使用方法，如图 3-18 所示。

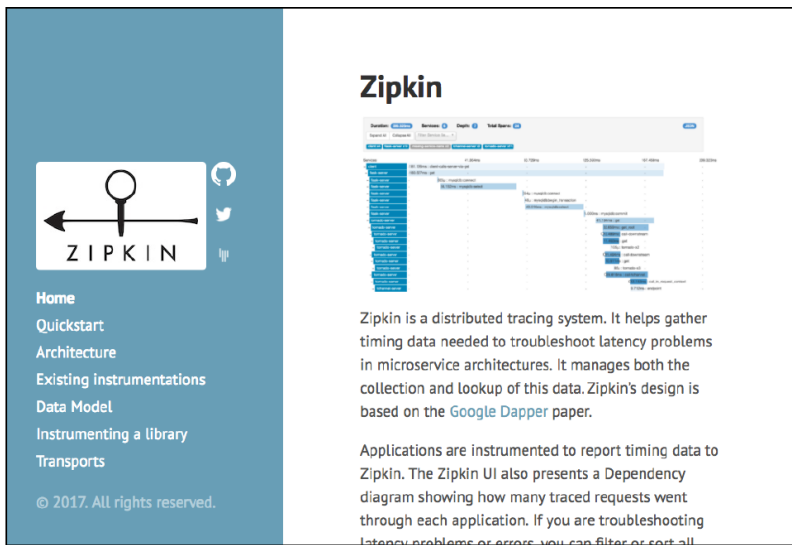


图 3-18 Zipkin 官网

Zipkin 官网: <http://zipkin.io/>。

如果想研究 Zipkin 的源码，可从它的 GitHub 站点获取。

Zipkin 源码: <https://github.com/openzipkin/zipkin/>。

在使用 Zipkin 之前，我们有必要学习它的几个核心概念。

1) Span

Zipkin 中提到的 Span 指的是调用一个组件所经历的一段过程，也就是说，从请求组件开始，直到组件响应为止，在这段过程中会花费一定的时间，这是一个时间跨度，所以我们形象地将其称为 Span。在 Zipkin 中，每个 Span 都带有一个可以唯一识别的 ID 号，我们称其为 Span ID。

2) Trace

Trace 指的是从客户端发出请求，直到完成整个内部调用的全部过程，我们将这个过程称为一次追踪，Trace 就是这次追踪过程。在 Zipkin 中，每个 Trace 也带有一个可以唯一识别的 ID 号，我们称其为 Trace ID。可见，一个 Trace 可能包含一个或多个 Span，反过来说，每个 Span 都能找到与它相对应的 Trace。由于组件之间会产生调用关系，那么每个 Span 也会出现依赖关系，因此每个 Span 都有对应的上级 Span，即 Parent Span，我们会用 Parent ID 来表示当前 Span 所依赖的 Parent Span。

3) Reporter

我们需要将 Span 与 Trace 所产生的追踪数据推送至 Zipkin 中，因此需要在相关的组件中安置一个客户端，它用于收集这些追踪数据并将它们报告给 Zipkin，我们将这个客户端称为 Reporter。Zipkin 官网以及开源社区提供了大量实现 Reporter 的代码库，我们可根据具体编程语言自行选择。

当熟悉了 Zipkin 的核心概念以后，我们再从它的系统架构上进行学习，如图 3-19 所示。

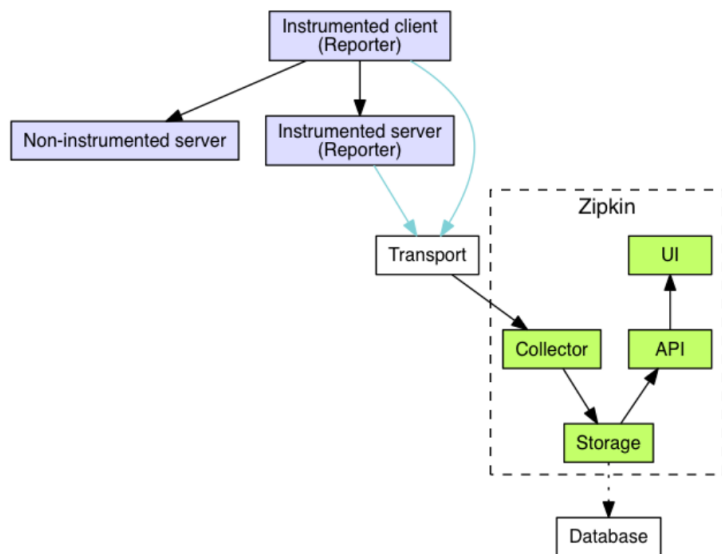


图 3-19 Zipkin 系统架构

只有被 Reporter 装配的（Instrumented）组件才能通过 Transport 向 Zipkin 发送数据，随后通过 Collector 进行数据收集，并通过 Storage 进行数据存储（可将数据持久化到 Database 中）。此后，可通过 API（Query Service）来查询 Storage 中的数据，并通过一个 UI（Web UI）来展示 Trace 与 Span 的调用链及其相关数据。

可以把 Transport 理解为一个数据传输方式，Zipkin 提供给了多种主要方式。最简单的情况下，可通过 HTTP 来传输数据；在并发量较高的情况下，可通过 Kafka 或 Scribe 来传输数据，可起到数据缓冲的作用，提高了整个调用追踪中心的吞吐率。

Kafka 是一个 Apache 开源的一款分布式消息系统，Scribe 是 Facebook 开源的一款日志收集系统，大家可通过它们的官网了解更多相关信息。

Kafka 官网：<http://kafka.apache.org/>。

Scribe 官网：<https://github.com/facebookarchive/scribe>。

下面，我们就开始进入 Zipkin 的内部来学习它的系统架构，不妨就它的入口 Collector 开始吧。

1) Collector

Collector 是一个收集数据的守护进程，当数据通过 Transport 进入 Collector 以后，将进行验证、存储、索引等步骤，以便 Zipkin 可以更加高效地获取这些数据。

2) Storage

默认情况下，Zipkin 的数据是放入内存中的，因此无法做到数据的持久化，也就是说，Zipkin 重启后数据就会被清空。我们可设置数据的存储方式，将其存入 MySQL、ElasticSearch 或 Cassandra 中。在数据量不是特别大的情况下，我们一般会考虑使用 MySQL 来存储数据。

3) Query Service

Query Service 实际上是一个返回 JSON 数据的 REST API，可通过该 API 获取存储在 Storage 中的数据。一般情况下，Query Service API 无须我们直接调用，而是仅对于 Zipkin 所提供的 Web 应用程序来使用。

4) Web UI

Web UI 就是 Zipkin 自带的 Web 应用程序，通过它可以查询并浏览 Storage 中存储的相关数据。需要说明的是，这个 Web UI 比较简单，启动后可直接访问，它并没有提供用户身份认证功能。一般情况下，我们仅在局域网中使用，不要对公网开放。

启动 Zipkin 最简单的方式莫过于 Docker 容器，我们可使用以下 docker run 命令来启动 Zipkin 容器。

```
docker run \  
-d \  
-p 9411:9411 \  
--name zipkin \  
openzipkin/zipkin
```

Zipkin 对外暴露 9411 端口，可通过 `http://localhost:9411/地址` 来访问 Zipkin Web UI，如图 3-20 所示。

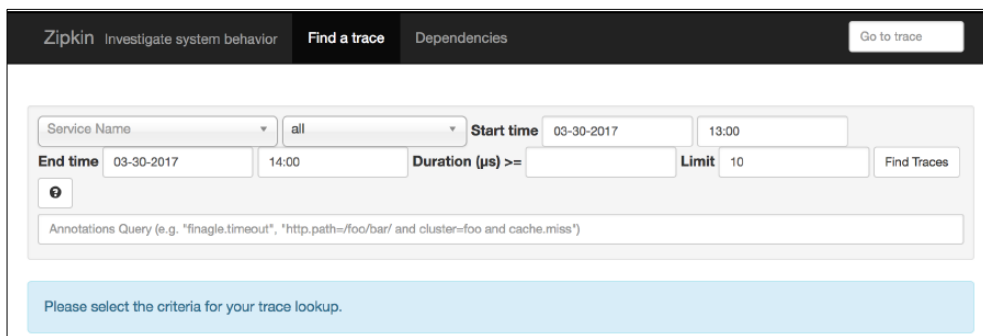


图 3-20 Zipkin Web UI 主界面

在 Zipkin Web UI 中，包含以下两个主要功能。

- (1) Find a trace: 查询相关的 Trace。
- (2) Dependencies: 查看每个组件（或服务）之间的调用依赖关系图。

此时在 Web UI 中看不到任何 Trace，因为我们还没有通过任何的 Zipkin 客户端产生相关追踪数据，随后我们将使用 Zipkin 的 Java 客户端 Brave 来收集并推送追踪数据，并将 Brave 集成到 Spring Boot 应用程序中。

Zipkin Java 客户端 (Brave): <https://github.com/openzipkin/brave>。

3.3.2 追踪微服务调用链

我们将开发 3 个基于 Spring Boot 的微服务，并有意识地让它们形成一个调用链关系，从而学习 Zipkin 的使用方法。

- (1) foo 服务：通过 REST API 方式来调用 bar 服务。
- (2) bar 服务：通过 REST API 方式来调用 hello 服务。
- (3) hello 服务：输出一个 “hello” 字符串。

简单情况下，我们可使用 Spring 自带的 `RestTemplate` 来实现 REST API 调用，首先在 Spring Context 中创建一个 `RestTemplate` 的 Spring Bean，然后将其注入到相应的 Controller 中即可。

下面，我们以 foo 服务为例，通过代码的方式加以说明。

首先，我们为 foo 服务创建一个 Spring Boot 应用程序类 `FooApplication`，代码如下：

```
package demo.msa.zipkin.foo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;

@SpringBootApplication
public class FooApplication {

    public static void main(String[] args) {
        SpringApplication.run(FooApplication.class, args);
    }

    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

`FooApplication` 类不仅带有 `@SpringBootApplication` 注解，还有一个 `main` 方法，此外我们还在该类中通过 `@Bean` 注解声明了一个 `RestTemplate` 的 Spring Bean，在 Spring Boot 应用程序启动后，将自动加载 `@Bean` 注解声明的对象，并将其放入 Spring Context 中，以便其他类进行注入并使用。

然后，我们创建一个 `FooController` 类，并在该类中注入并使用 `FooApplication` 类中声明的 `RestTemplate` 对象。

```
package demo.msa.zipkin.foo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
```

```
import org.springframework.web.client.RestTemplate;

import java.util.Random;

@RestController
public class FooController {

    @Autowired
    private RestTemplate restTemplate;

    @GetMapping("/foo")
    public String foo() {
        try {
            Thread.sleep(new Random().nextInt(1000));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return restTemplate.getForObject("http://localhost:8082/bar", String.class);
    }
}
```

此时，我们使用了`@Autowired`注解将`RestTemplate`对象通过成员变量注入到`FooController`类中。实际上，`FooController`中带有`@RestController`注解，带有该注解的类将被Spring扫描，随后通过反射的方式创建`FooController`对象，并将Spring Context中的`RestTemplate`对象注入到`FooController`类的同类型成员变量中。我们在`foo`方法中使用了一段随机数休眠方式，让应用程序的主线程执行到此处时可以休眠一段时间，这样做只是为了模拟调用“GET /foo”时需要花费一段相对较长的时间，以便我们更清晰地观察调用链追踪轨迹。在`foo`方法返回之前，我们使用`RestTemplate`对象的`getForObject`方法发送了一个GET请求，调用`bar`服务的REST API。

为了便于操作，我们将`foo`、`bar`、`hello`三个服务分别部署到8081、8082、8080端口，我们可在`application.properties`文件中自行配置，下面是`foo`服务的端口配置。

```
server.port=8081
```

类似地，我们在`bar`服务中也使用以上方式来调用`hello`服务。

```
package demo.msa.zipkin.bar;
```

```
import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

import java.util.Random;

@RestController
public class BarController {

    @Autowired
    private RestTemplate restTemplate;

    @GetMapping("/bar")
    public String bar() {
        try {
            Thread.sleep(new Random().nextInt(1000));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return restTemplate.getForObject("http://localhost:8080/hello",
String.class);
    }
}
```

在 hello 服务中，我们只是简单地输出了一个“hello”字符串。

```
package demo.msa.zipkin.hello;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.Random;

@RestController
public class HelloController {

    @GetMapping("/hello")
```

```
public String hello() {  
    try {  
        Thread.sleep(new Random().nextInt(1000));  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
    return "hello";  
}
```

我们的目标是将以上三个服务在运行时都进行调用链追踪，以 foo 服务为例，我们需要在 application.properties 文件中添加关于 Zipkin 的相关配置。

```
zipkin.endpoint=http://localhost:9411/api/v1/spans  
zipkin.service=foo
```

其中，zipkin.endpoint 配置项用于指定 Zipkin 的 API 端点，当 Zipkin 启动后，该端点地址就固定下来了。zipkin.service 配置项用于指定当前服务的名称，用于区分当前服务对应哪个 Span。bar 服务与 hello 服务关于 Zipkin 的配置与以上 foo 服务类似，唯一不同在于 zipkin.service 配置项。

需要说明的是，以上这两项配置并非 Spring Boot 所提供的，而是我们自定义的，这些配置将通过一个公共组件来解析，并通过该组件与 Zipkin 进行通信，将每个应用组件所产生的追踪数据收集并推送至 Zipkin 中。

接下来我们需要做的是，开发一个封装 Brave 的公共组件 zipkin-common，该组件用于以上 foo、bar、hello 等需要被追踪的应用组件。

创建一个 Maven 项目，该项目的 Maven 三坐标如下所示。

- groupId: demo.msa;
- artifactId: zipkin-common;
- version: 1.0.0。

由于我们在需要被追踪的服务中定义了两个 Zipkin 的配置项，因此我们第一步要做的是，在 zipkin-common 组件中定义一个配置属性类，用于封装以上配置项。

```
package demo.msa.zipkin.common;  
  
import org.springframework.boot.context.properties.ConfigurationProperties;  
import org.springframework.stereotype.Component;
```

```
@Component
@ConfigurationProperties("zipkin")
public class ZipkinProperties {

    private String endpoint;

    private String service;

    public String getEndpoint() {
        return endpoint;
    }

    public void setEndpoint(String endpoint) {
        this.endpoint = endpoint;
    }

    public String getService() {
        return service;
    }

    public void setService(String service) {
        this.service = service;
    }
}
```

我们编写了一个 `ZipkinProperties` 类，该类包含 `endpoint` 与 `service` 两个成员变量，以及它们所对应的 `getter` 与 `setter` 方法，它是一个标准的 `JavaBean` 类。由于 `ZipkinProperties` 类需要被 `Spring` 容器扫描并加载，因此需要在类上添加 `@Component` 注解。同时，当 `Spring Boot` 扫描到该类，发现它带有 `@ConfigurationProperties` 注解，此时将加载 `application.properties` 文件，并将前缀为 `zipkin` 的配置项分别加载到对应名称的成员变量中，此时用到了 `Java` 的反射技术，调用了对应的 `setter` 方法进行成员变量的初始化。后续我们可将 `ZipkinProperties` 类注入到其他类中，并随时通过 `getter` 方法来访问对应的成员变量，它们实际上就是 `application.properties` 文件中所包含的 `zipkin` 前缀的相关配置项。

我们需要在 `pom.xml` 文件中添加 `Spring Boot` 的 `Maven` 依赖，它将自动添加所对应的 `Spring` 依赖。由于 `Spring Boot` 依赖一般情况下由应用方（例如：`foo`、`bar`、`hello`）提供，因此我们只需将 `Spring Boot` 依赖设置为 `provided` 即可。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot</artifactId>
  <version>1.5.2.RELEASE</version>
  <scope>provided</scope>
</dependency>
```

下面我们做的这件事情非常重要，需要对底层 Spring MVC 的运行过程进行一些扩展。当我们使用 RestTemplate 发送 REST API 时，需将 Brave 的 BraveClientHttpRequestInterceptor 拦截器添加到 RestTemplate 的拦截器调用链中，这样有助于收集相关请求数据并发送到 Zipkin 中。此外，还需将 Brave 的 ServletHandlerInterceptor 拦截器注册到 Spring MVC 拦截器框架中，以确保服务端可收集相关响应数据到 Zipkin 中。这样一来，请求数据和响应数据都能通过 Brave 发送到 Zipkin 中了。

紧接着，我们需要继续在 pom.xml 文件中添加 Spring MVC 与 Brave 相关的 Maven 依赖。由于 Spring MVC 依赖一般情况下由应用方提供，因此我们只需将 Spring Boot 依赖设置为 provided 即可。简单情况下，我们可使用 HTTP 作为 Zipkin 的 Reporter 传输方式，因此还需要添加 Zipkin Reporter 基于 OkHttp 的 Maven 依赖。

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>4.3.7.RELEASE</version>
  <scope>provided</scope>
</dependency>

<dependency>
  <groupId>io.zipkin.brave</groupId>
  <artifactId>brave-spring-web-servlet-interceptor</artifactId>
  <version>4.0.6</version>
</dependency>

<dependency>
  <groupId>io.zipkin.brave</groupId>
  <artifactId>brave-spring-resttemplate-interceptors</artifactId>
  <version>4.0.6</version>
</dependency>
```

```

<dependency>
  <groupId>io.zipkin.brave</groupId>
  <artifactId>brave-mysql</artifactId>
  <version>4.0.6</version>
</dependency>

<dependency>
  <groupId>io.zipkin.reporter</groupId>
  <artifactId>zipkin-sender-okhttp3</artifactId>
  <version>0.6.12</version>
</dependency>

```

下面我们需要编写的是一个 Spring 的配置类，该类需使用 `@Configuration` 注解来定义，并可在该类中使用 `@Bean` 注解来定义所需的相关 Spring Bean，当然也可通过 `@Autowired` 注解来注入所需的 Spring Bean，有些 Spring Bean 在其他 Spring Context 中创建（比如：在应用方的代码中，或者在其他依赖包中），因此需要使用 `@Import` 注解将这些 Spring Bean 所对应的类加以描述。

首先，我们需要编写一个 `ZipkinConfiguration` 类，并在该类中创建一些 Brave 相关的 Spring Bean。

```

@Configuration
public class ZipkinConfiguration {

    @Autowired
    private ZipkinProperties zipkinProperties;

    @Bean
    public Sender sender() {
        return OkHttpSender.create(zipkinProperties.getEndpoint());
    }

    @Bean
    public Reporter<Span> reporter() {
        return AsyncReporter.builder(sender()).build();
    }

    @Bean

```

```

    public Brave brave() {
        return new Brave.Builder(zipkinProperties.getService()).reporter
(reporter()).build();
    }

    @Bean
    public SpanNameProvider spanNameProvider() {
        return new SpanNameProvider() {
            @Override
            public String spanName(HttpServletRequest httpRequest) {
                return String.format(
                    "%s %s",
                    httpRequest.getHttpMethod(),
                    httpRequest.getUri().getPath()
                );
            }
        };
    }
}

```

我们首先注入了 `ZipkinProperties` 类，并通过该类所提供的 `getter` 方法来获取相关 `endpoint` 与 `service` 成员变量中的值。`Sender` 是基于 `OkHttp` 来创建的，此时需要传入 `Zipkin` 的 `endpoint` 参数。`Reporter` 需要根据现有的 `Sender` 来构建，此时将通过 `AsyncReporter` 类使用异步方式将追踪数据传送到 `Zipkin` 中，此时也能返回一个 `LoggingReporter` 对象（`return new LoggingReporter()`），表示将追踪数据输出到日志中，但这明显不是我们最终想要的效果。`Brave` 对象需要传入被追踪的服务名称，以及通过现有的 `Reporter` 对象来构建。还需要创建一个 `SpanNameProvider`，用于生成具体的 `Span` 名称，可返回一个 `DefaultSpanNameProvider` 对象（`return new DefaultSpanNameProvider()`），将默认使用 `HTTP` 方法作为 `Span` 名称，也能自行定义 `Span` 名称的命名规则，此时我们自定义了 `Span` 名称的命名规则，选择了“`HTTP` 方法 + `URI` 路径”的方式作为 `Span` 名称。

然后，我们要做的是在 `ZipkinConfiguration` 类中注入一些后面需要用到的 `Spring Bean`。

```

@Configuration
@Import({RestTemplate.class, BraveClientHttpRequestInterceptor.class,
ServletHandlerInterceptor.class})
public class ZipkinConfiguration {

```



```

...

@Autowired
private RestTemplate restTemplate;

@Autowired
private BraveClientHttpRequestInterceptor clientInterceptor;

@Autowired
private ServletHandlerInterceptor serverInterceptor;
}

```

由于这些 Spring Bean 都是其他 Spring Context 中创建的，其中 RestTemplate 来自应用方（即 foo、bar 等服务），BraveClientHttpRequestInterceptor 与 ServletHandlerInterceptor 来自 Brave 相关 Maven 组件，因此需要使用 @Import 注解将这些类导入进来，这样才能使用 @Autowired 注解将这些类所对应的对象注入到 Spring Context 中，从而正常使用它们。

最后，我们将使用以上注入的三个 Spring Bean，以完成我们最终的目标。

```

...

public class ZipkinConfiguration extends WebMvcConfigurerAdapter {

    ...

    @PostConstruct
    public void init() {
        List<ClientHttpRequestInterceptor> interceptors =
restTemplate.getInterceptors();
        interceptors.add(clientInterceptor);
        restTemplate.setInterceptors(interceptors);
    }

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(serverInterceptor);
    }
}

```

我们需要将 `ZipkinConfiguration` 继承于 `WebMvcConfigurerAdapter` 父类，它是 Spring MVC 的配置适配器，我们可通过覆盖父类的方法来扩展 Spring MVC 的行为。此时我们覆盖父类的 `addInterceptors` 方法，将 `serverInterceptor` (`ServletHandlerInterceptor`) 添加到 Spring MVC 拦截器调用链中，从而收集服务端响应追踪数据。此外，还通过编写一个带有 `@PostConstruct` 注解的初始化方法，将 `clientInterceptor` (`BraveClientHttpRequestInterceptor`) 添加到 `RestTemplate` 的拦截器调用链中，从而收集客户端请求追踪数据。可见 Spring MVC 与 `RestTemplate` 都提供了相应的拦截器调用链机制，我们只需扩展 `WebMvcConfigurerAdapter` 父类就能修改这些拦截器的调用过程，以便在调用过程中添加我们需要的操作。

`ZipkinConfiguration` 稍微有些复杂，以下是它的完整代码，便于大家阅读与实践。

```
package demo.msa.zipkin.common;

import com.github.kristofa.brave.Brave;
import com.github.kristofa.brave.http.HttpRequest;
import com.github.kristofa.brave.http.SpanNameProvider;
import com.github.kristofa.brave.spring.BraveClientHttpRequestInterceptor;
import com.github.kristofa.brave.spring.ServletHandlerInterceptor;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
import org.springframework.http.client.ClientHttpRequestInterceptor;
import org.springframework.web.client.RestTemplate;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.

WebMvcConfigurerAdapter;
import zipkin.Span;
import zipkin.reporter.AsyncReporter;
import zipkin.reporter.Reporter;
import zipkin.reporter.Sender;
import zipkin.reporter.okhttp3.OkHttpSender;

import javax.annotation.PostConstruct;
import java.util.List;

@Configuration
@Import({RestTemplate.class, BraveClientHttpRequestInterceptor.class,
```

```
ServletHandlerInterceptor.class}}

public class ZipkinConfiguration extends WebMvcConfigurerAdapter {

    @Autowired
    private ZipkinProperties zipkinProperties;

    @Bean
    public Sender sender() {
        return OkHttpSender.create(zipkinProperties.getEndpoint());
    }

    @Bean
    public Reporter<Span> reporter() {
        return AsyncReporter.builder(sender()).build();
    }

    @Bean
    public Brave brave() {
        return new Brave.Builder(zipkinProperties.getService()).reporter
(zipkinProperties.getReporter()).build();
    }

    @Bean
    public SpanNameProvider spanNameProvider() {
        return new SpanNameProvider() {
            @Override
            public String spanName(HttpServletRequest httpServletRequest) {
                return String.format(
                    "%s %s",
                    httpServletRequest.getMethod(),
                    httpServletRequest.getUri().getPath()
                );
            }
        };
    }

    @Autowired
    private RestTemplate restTemplate;
```

```
@Autowired
private BraveClientHttpRequestInterceptor clientInterceptor;

@Autowired
private ServletHandlerInterceptor serverInterceptor;

@PostConstruct
public void init() {
    List<ClientHttpRequestInterceptor> interceptors =
restTemplate.getInterceptors();
    interceptors.add(clientInterceptor);
    restTemplate.setInterceptors(interceptors);
}

@Override
public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(serverInterceptor);
}
}
```

至此，zipkin-common 组件就开发完毕了，下面我们需要将其应用到 foo、bar、hello 服务（应用方）中，只需在这些服务中添加以下 Maven 配置即可。

```
<dependency>
  <groupId>demo.msa</groupId>
  <artifactId>zipkin-common</artifactId>
  <version>1.0.0</version>
</dependency>
```

此外，还需对应用方加以配置，才能加载以上 zipkin-common 组件。

下面以 foo 服务为例加以说明。

需要在 foo 服务的应用程序启动类 FooApplication 中进行如下调整：

```
package demo.msa.zipkin.foo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
```

```
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;

@SpringBootApplication(scanBasePackages = "demo.msa.zipkin")
public class FooApplication {

    public static void main(String[] args) {
        SpringApplication.run(FooApplication.class, args);
    }

    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

此时在`@SpringBootApplication`注解中添加了`scanBasePackages`属性，它表示需要扫描的基础包名。默认情况下，只会扫描`FooApplication`类所在的包名，即`demo.msa.zipkin.foo`包中的类。由于`zipkin-common`组件中提供的类均在`demo.msa.zipkin.common`包中，如果想扫描`zipkin-common`组件，那么必须扩大包的扫描范围，此时将`scanBasePackages`设置为`demo.msa.zipkin`，可以保证既能扫描当前应用程序包（`demo.msa.zipkin.foo`），又能扫描`zipkin-common`组件所在的包（`demo.msa.zipkin.common`）。此外，也在`FooApplication`类中通过`@Bean`注解声明了一个`RestTemplate`的Spring Bean。应用程序启动后，Spring将扫描带有`@Bean`注解的方法，通过反射的方式调用这些方法，从而创建对应的Bean对象，最终将这些对象放入Spring Context中，以便后续依赖注入，这是Spring框架的核心价值。

分别运行`foo`、`bar`、`hello`服务，并在浏览器中发送几次`http://localhost:8081/foo`请求，随后打开Zipkin Web UI并点击Find Traces按钮，可看到以下Trace信息，如图3-21所示。

该界面分为上下两部分，上部分是查询表单，下部分是查询结果。查询表单中的第一个下拉框列出了所有Service名称，我们可查看某个Service的调用链情况。查询表单中的第二个下拉框列出了所有Span名称，用于过滤某个Service中的特定Span。我们也能通过输入Start time与End time来过滤请求时间，通过Duration来过滤调用持续时长，通过Limit来限制查询结果条数，甚至能通过输入一段Zipkin内置的Annotation Query做一些更高级别的查询。对于查询出来的结果，我们可通过不同的优先规则对查询结果进行排序，默认是Longest First（最长优先），也可以设置为Newest First（最新优先），这样最新产生的Trace信息就会根据请求时间倒序排列了，即最新的Trace放在最上面。可点击JSON按钮，在弹出的对话框中查看当前查询结果所对应的JSON数据。

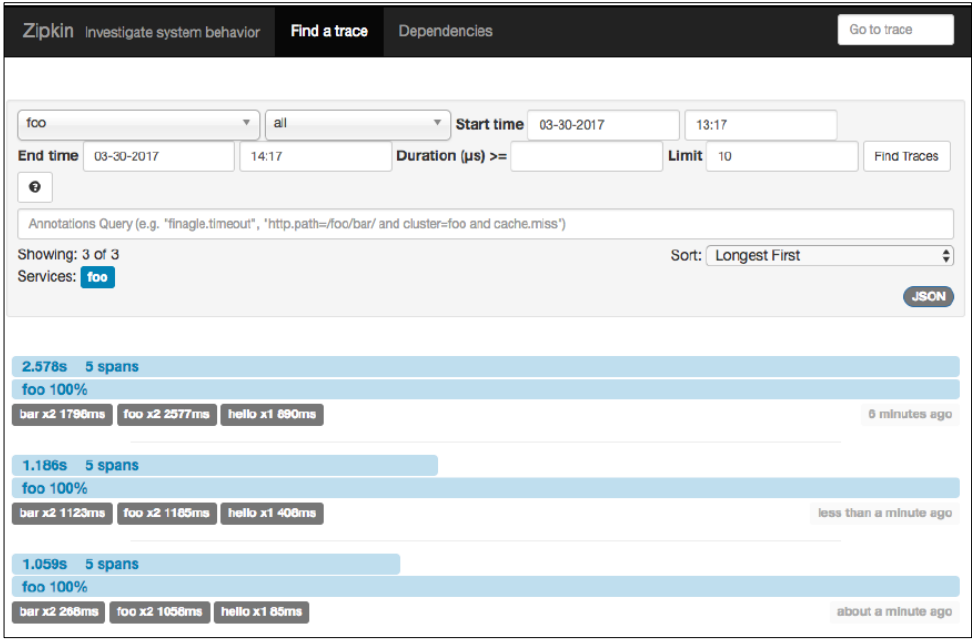


图 3-21 Trace 查询

在查询结果中用水平柱状图表示每次 Trace 的调用概要信息，我们点击其中一条 Trace，可查看该 Trace 的调用链追踪信息，如图 3-22 所示。

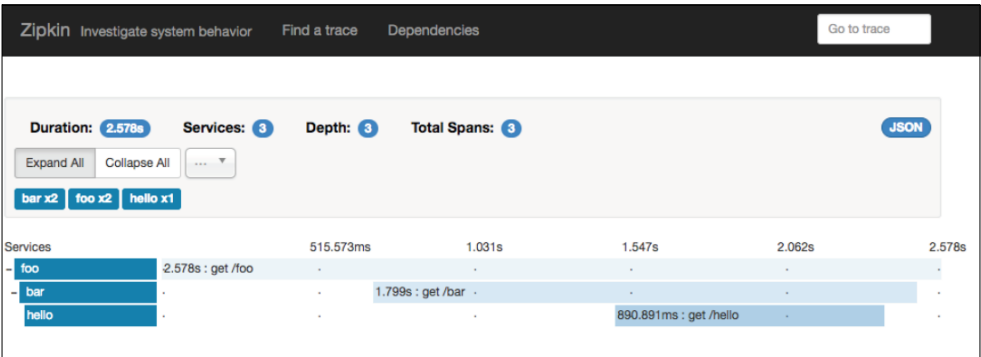


图 3-22 Trace 与 Span

该界面同样也分为上下两部分，上部分是调用链的概要信息，包括 Durations（调用所产生的持续时长）、Services（调用所涉及的服务数量）、Depth（调用深度，即调用了多少次）、Total Spans（调用所包含的 Span 总数）。可通过点击 Expand All 或 Collapse All 来展开或折叠下方的调用链追踪图。可点击 JSON 按钮，在弹出的对话框中查看当前调用链所对应的 JSON 数据。

以上包含 foo、bar、hello 三个 Span，我们点击 hello 这个 Span，可在弹出的对话框中通过表格的方式查看该 Span 的详细信息，如图 3-23 所示。

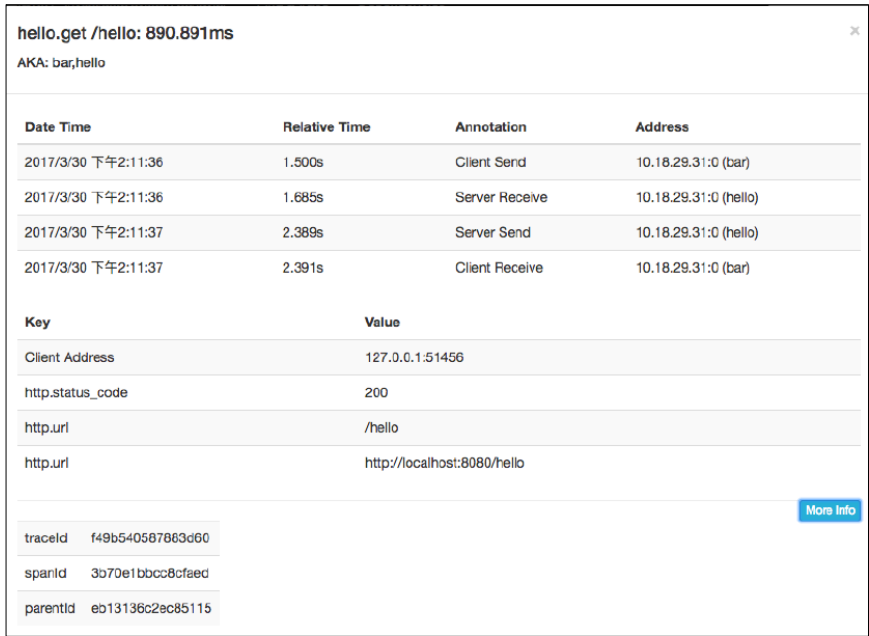


图 3-23 Span 详细信息

每个 Span 包含上中下三个表格数据，上方表格表示客户端与服务端的请求与响应时间，包含 4 种标注。

- (1) Client Send：表示客户端发送数据所花费的时间。
- (2) Server Receive：表示服务端接收数据所花费的时间。
- (3) Server Send：表示服务端发送数据所花费的时间。
- (4) Client Receive：表示客户端接收数据所花费的时间。

可见，1 次 Span 包括以上 4 个过程，每个过程都花费一定的时间开销。对于 hello 而言，bar 是客户端，hello 是服务端。

中间表格表示客户端相关信息，包括 Client Address（客户端 IP 与端口）、http.status_code（客户端 HTTP 状态码）、http.url（客户端 HTTP 请求地址）。

下方表格包含 3 个重要的 ID 信息，它们都用 16 位的十六进制数据来表示。

- (1) traceId：Span 所对应的 Trace ID。
- (2) spanId：自身的 Span ID。

(3) parentId: 当前 Span 所依赖的上级 Span ID。

此时,我们也能在 Dependencies 中查看当前一段时间内的调用依赖关系图,如图 3-24 所示。

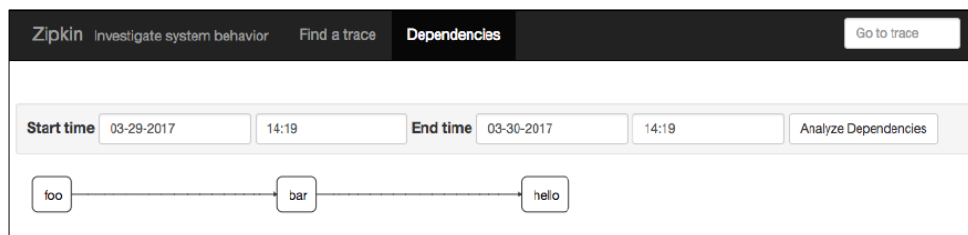


图 3-24 调用依赖关系图

从以上调用依赖关系图中,可清晰地看到每个服务的调用关系,我们通过 foo 服务去调用 bar 服务,通过 bar 服务去调用 hello 服务。对于复杂的调用关系而言,这个图会更有价值。

如果在 hello 服务中需要对数据库进行操作,那么应该怎样追踪数据库的调用呢?我们下面就来体验 Zipkin 对数据库调用链追踪的特性。

3.3.3 追踪数据库调用链

Brave 当前仅对 MySQL 数据库提供了直接的支持,稍微对以上 zipkin-common 组件加以扩展,就能对 MySQL 数据库的调用进行追踪。

首先,在 pom.xml 文件中添加以下 Maven 依赖。

```
<dependency>
  <groupId>io.zipkin.brave</groupId>
  <artifactId>brave-mysql</artifactId>
  <version>4.0.6</version>
</dependency>
```

以上添加了一个 brave-mysql 组件,它也是 Brave 内置的组件之一。

然后,在 ZipkinConfiguration 类中声明以下 Spring Bean。

```
@Bean
public MySQLStatementInterceptorManagementBean
mysqlStatementInterceptorManagementBean() {
    return new MySQLStatementInterceptorManagementBean(brave().clientTracer());
}
```


brave-mysql 组件将使用 `MySQLStatementInterceptorManagementBean` 对象对所有发送至 MySQL 数据库的 SQL 语句进行拦截, 创建该对象时需要传入当前 Brave 对象的 `ClientTracer` 对象。

对 `zipkin-common` 的扩展到此为止, 下面我们需要对应用方加以配置, 以支持 MySQL 调用追踪特性。

下面以 `hello` 服务为例加以说明。

首先, 在 `pom.xml` 文件中添加以下 Maven 依赖。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>

<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <scope>runtime</scope>
</dependency>
```

使用 `spring-boot-starter-jdbc` 组件是为了在当前应用程序中能直接使用 `JdbcTemplate` 对象对所指定的数据库执行 JDBC 操作。使用 `mysql-connector-java` 组件开启了当前应用程序对 MySQL 数据库的支持。

然后, 在 `application.properties` 文件中添加以下配置项。

```
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost/demo?useSSL=false\
  &statementInterceptors=com.github.kristofa.brave.mysql.MySQLStatementI
nterceptor\
  &zipkinServiceName=mysql
spring.datasource.username=root
spring.datasource.password=root
```

使用 `spring.datasource` 开头的配置项来指定目标数据源, 即 MySQL 数据库。需要注意的是 `spring.datasource.url` 配置项, 其中不仅指定了 MySQL 的连接 URL, 还在 URL 中添加了 `statementInterceptors` 与 `zipkinServiceName` 参数。前者表示拦截 MySQL 数据库 SQL 语句的拦截器, 即 `com.github.kristofa.brave.mysql.MySQLStatementInterceptor`; 后者表示在 Zipkin 中对应的

服务名称，它也是 MySQL 对应的 Span 名称，该名称的默认值是 “mysql- $\{database\}$ ”，即当前案例中的 mysql-demo，此时我们将其命名为 mysql。

然后，可在 HelloController 类中注入 JdbcTemplate 来访问数据库。

```
package demo.msa.zipkin.hello;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    @GetMapping("/hello")
    public String hello() {
        String name = jdbcTemplate.queryForObject(
            "SELECT name FROM customer WHERE id = 1",
            String.class
        );
        return "hello " + name;
    }
}
```

此时通过 @Autowired 注解注入的 JdbcTemplate 对象实际上是由 Spring Boot 创建的，因为此时我们在 Maven 中引入了 spring-boot-starter-jdbc 组件，并在 application.properties 文件中开启了 spring.datasource 的相关配置，那么 Spring Boot 就能自动创建 JdbcTemplate 对象，此时我们才能成功将其注入进来，并在相应的逻辑代码中调用 JdbcTemplate 对象的方法来操作数据库。此时我们只是通过某个 ID 号，在数据库的指定表中查询出一条数据，并将查询结果放入方法返回值中，通过这种简单的方式来体验 Zipkin 对数据库的调用追踪能力。

现在应用程序已经准备完毕，最后我们通过 Docker 来启动一个 MySQL 容器。

```
docker run \
-d \
```

```
-p 3306:3306 \
-v ~/mysql/data:/var/lib/mysql \
-v ~/mysql/conf:/etc/mysql/conf.d \
-e MYSQL_ROOT_PASSWORD=root \
--name mysql \
mysql
```

这里将容器中 MySQL 的数据目录与配置目录分别映射到宿主机中（-v），并通过环境变量指定了一个 root 用户的密码（-e）。

启动 MySQL 完毕后，可再运行一个 MySQL 容器作为命令行客户端来初始化相应的数据。

```
docker run \
-it \
--rm \
--link mysql:mysql \
mysql \
mysql -hmysql -uroot -p
```

进入 MySQL 客户端后，我们需要创建两个数据库，第一个数据库（zipkin）用于存储 Zipkin 所收集的追踪数据，第二个数据库（demo）用于存储当前示例中的业务数据。

Brave 的 GitHub 站点上已给出了 Zipkin 数据库的创建脚本，可通过以下地址获取。

Zipkin MySQL 数据库脚本：<https://github.com/openzipkin/zipkin/blob/master/zipkin-storage/mysql/src/main/resources/mysql.sql>。

首先需要创建一个名为 zipkin 的数据库，然后通过 MySQL 客户端执行以上脚本，将生成以下 3 张表。

- （1）zipkin_spans：用于存放每个 Span 信息。
- （2）zipkin_annotations：用于存放每个 Annotation 信息。
- （3）zipkin_dependencies：用于存放 Span 之间的依赖关系。

此外，我们还需创建第二个数据库，它是 hello 服务需要连接的数据库，以下是该数据库的初始化脚本。

```
CREATE DATABASE `demo`;

CREATE TABLE `customer` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
```

```
`name` varchar(100) DEFAULT NULL,  
PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;  
  
INSERT INTO `customer` (`id`, `name`) VALUES (1, 'jack');
```

当 MySQL 容器以及数据准备完成后，我们还需要重新启动一个 Zipkin 容器，将其连接到 MySQL 容器上。

```
docker run \  
-d \  
-p 9411:9411 \  
--link mysql:mysql \  
-e STORAGE_TYPE=mysql \  
-e MYSQL_HOST=mysql \  
-e MYSQL_TCP_PORT=3306 \  
-e MYSQL_DB=zipkin \  
-e MYSQL_USER=root \  
-e MYSQL_PASS=root \  
--name zipkin \  
openzipkin/zipkin
```

启动 Zipkin 容器时需要通过 Docker Link 方式连接到 MySQL 容器，并通过一系列环境变量来传入 Zipkin 所需的 MySQL 相关配置。

重启 hello 服务，在浏览器中发送 `http://localhost:8080/hello` 请求，不出意外的话，应该可以看到一个“hello jack”字符串被输出到浏览器中。

随后，可在 Zipkin Web UI 中查询调用以上服务所产生的 Trace 信息，如图 3-25 所示。

从此时的 Trace 中可以看到，当调用了一个 hello 服务后，相继调用了多次 mysql 服务，可点击每个 mysql 服务查看对应的详细信息。如果使用鼠标圈选相关的时间片段，将放大指定时间片段以内的 Trace 调用链轨迹。

如果我们在浏览器中再次发送 `http://localhost:8080/hello` 请求，将在 Zipkin Web UI 中看到不一样的 Trace 信息，如图 3-26 所示。

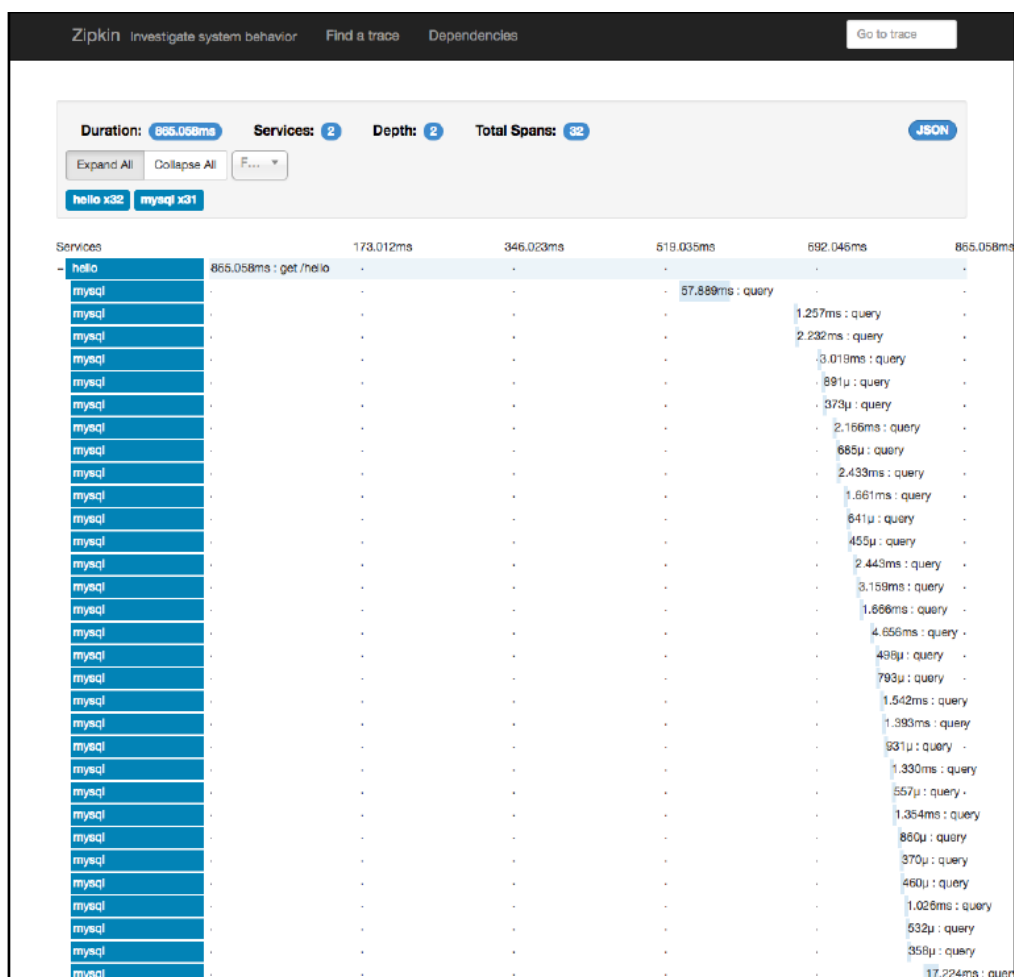


图 3-25 首次查询 MySQL 的调用链追踪

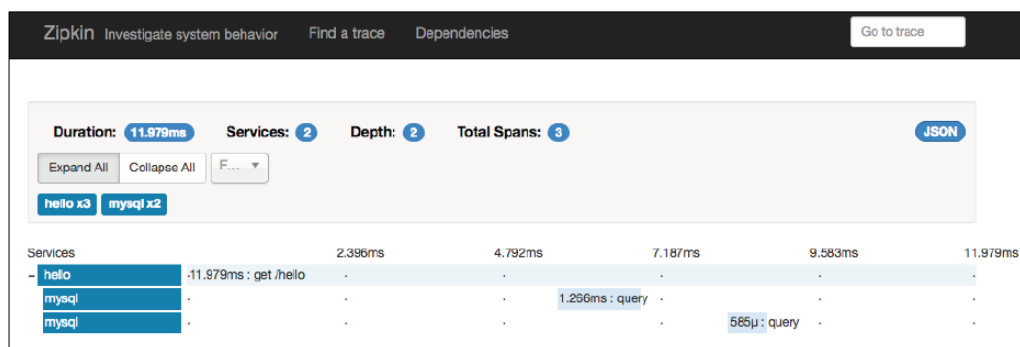


图 3-26 再次查询 MySQL 的调用链追踪

此时只看到两次 `mysql` 调用，分别点击以上两个 `mysql` 服务，在弹出的对话框中看到一个名为 `sql.query` 的 `key` 对应一个 SQL 语句，它就是 MySQL 实际执行的 SQL 语句。第一次 `mysql` 调用所执行的 SQL 语句是 `SELECT 1`，表示尝试连接 MySQL 数据库，判断当前是否可正常连接，第二次 `mysql` 调用所执行的 SQL 语句才是我们实际业务所发生的操作。可通过以上方法分析 MySQL 调用过程中相对较慢的 SQL 语句。

以上我们体验了使用 Brave 收集 Spring 与 MySQL 的追踪数据的方法，对于 Java 应用程序而言，Brave 官方还支持与其他框架的集成能力，比如：CXF、gRPC、Jersey、P6Spy、RESTEasy、Spark 等。如果大家觉得这些还不够的话，不妨使用 Brave 框架自行扩展。对于其他非 Java 的编程语言的技术，可使用 Zipkin 官方与开源社区所提供的客户端开发包，当然也能参考这些实现并自行开发。

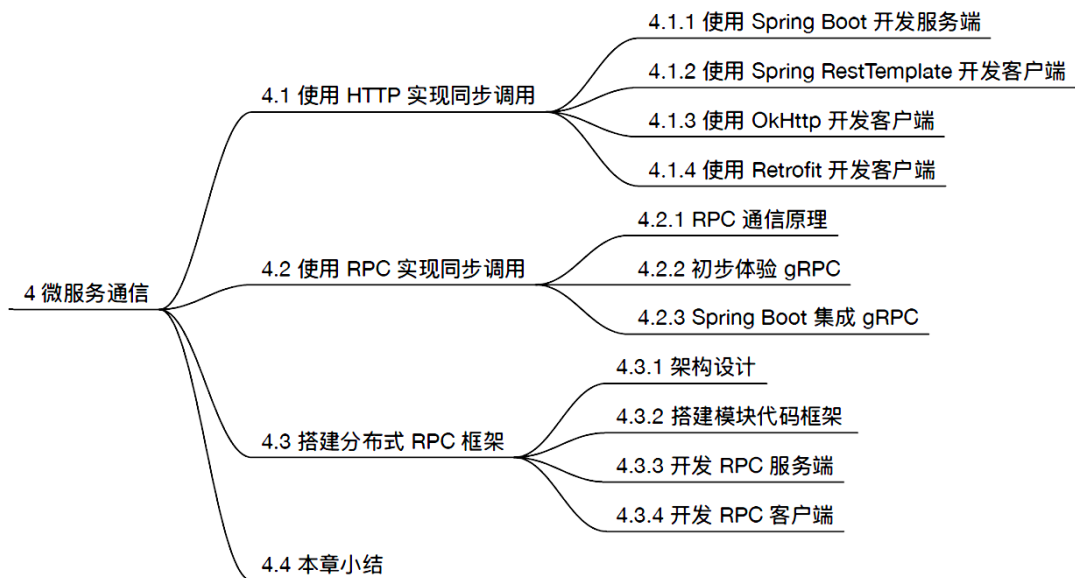
3.4 本章小结

本章的视角放在微服务监控方面，此外也讨论了服务调用链追踪问题，我们希望微服务的监控和追踪都能通过图形化的方式进行展现。首先我们学习了 Spring Boot 应用程序自带的监控特性，接着介绍了 Spring Boot Admin 开源监控系统的使用方法，这些工具我们可以很容易地使用起来，但是仍然存在一些弊端。随后我们集成了 InfluxDB、cAdvisor、Grafana 等开源工具，并将它们整合为一款微服务的“监控中心”，通过观察监控中心上的数据变化情况，我们可以随时观察当前的微服务运行状态。最后我们学习了 Zipkin 工具的使用方法，它可扮演微服务的“追踪中心”的角色，我们可以清晰地观察到调用链的关联关系与调用开销。

下一章我们将沿着调用链继续探险，希望能够使用合适的技术来解决服务之间的通信问题。

4 chapter

第 4 章 微服务通信



4.1 使用 HTTP 实现同步调用

当一个服务调用另一个服务时，最简单的方式就是基于 HTTP 进行同步调用。所谓同步调用，意味着在调用目标服务时，调用方始终处于等待调用返回的状态，此时调用方无法做任何事情。我们想要实现 HTTP 同步调用其实很简单，只需目标服务对外暴露相关 HTTP 请求地址（包括域名或 IP、端口、路径等），调用方就能通过该 HTTP 请求地址调用目标服务。一般情况下，我们可使用 RESTful 作为开发规范，将服务对外暴露的 HTTP 调用方式暴露为 REST API。我们目前所探讨的微服务对外就提供了一系列 REST API，只是它们都是供服务网关来调用的，而不存在任何服务之间的调用。当然，我们也可以通过 HTTP 方式实现服务之间的同步调用。

下面，我们首先使用 Spring Boot 开发一个服务端，紧接着使用 Spring RestTemplate 作为客户端来实现 HTTP 同步调用。最后我们也会探索一些流行的 HTTP 客户端工具（例如，OkHttp、Retrofit 等），来实现 HTTP 同步调用，通过对比来选择最适合我们自己的同步调用工具。

4.1.1 使用 Spring Boot 开发服务端

使用 Spring Boot 开发服务端是一件非常简单的事情，只需按照以下三个步骤操作即可完成。

第一步：添加 Spring Boot 相关 Maven 依赖。

在 pom.xml 文件中添加如下 Spring Boot 的 Maven 依赖。

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.5.2.RELEASE</version>
</parent>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
</dependencies>
```

首先需要继承一个 Spring Boot 所提供的 parent 依赖 spring-boot-starter-parent，然后在 dependencies 中添加所需的依赖，目前我们添加了 spring-boot-starter-web 依赖，因为我们想要开发的是一个 Java Web 应用，它将对外暴露相关 HTTP 请求地址。

第二步：编写服务定义并完成服务逻辑实现。

在 `demo.msa.hello.server` 包中创建以下 Java 类：

```
package demo.msa.hello.server;

import org.springframework.web.bind.annotation.*;

@RestController
public class SayController {

    @GetMapping("/say1")
    public String say1(@RequestParam("name") String name) {
        return "hello " + name;
    }

    @GetMapping("/say2/{name}")
    public String say2(@PathVariable("name") String name) {
        return "hello " + name;
    }

    @GetMapping("/say3")
    public String say3(@RequestHeader("name") String name) {
        return "hello " + name;
    }
}
```

我们创建了一个名为 `SayController` 的类，并在该类上使用 `@RestController` 注解将该类定义为一个具备 REST API 能力的 Controller。随后我们在该 Controller 类中的特定方法上使用 `@GetMapping` 注解来定义 GET 类型的 HTTP 请求，服务相关逻辑实现均在该注解定义方法的内部来填充。以上我们定义了三个 REST API，并通过不同方式将参数传入方法中。

- `@RequestParam`：表示从 HTTP 请求参数中获取指定名称的参数。
- `@PathVariable`：表示从 HTTP 请求路径中获取指定名称的参数（路径中需表明该参数的具体位置）。
- `@RequestHeader`：表示从 HTTP 请求头中获取指定名称的参数。

后面我们将通过 HTTP 客户端工具来调用以上定义的 REST API，此时仅为服务端实现。

第三步：编写 Spring Boot 应用启动程序。

在 demo.msa.hello.server 包中创建以下 Java 类：

```
package demo.msa.hello.server;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class HelloServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(HelloServerApplication.class, args);
    }
}
```

我们创建了一个名为 HelloServerApplication 的类，并在该类上使用 @SpringBootApplication 注解将该类定义为一个 Spring Boot 应用程序启动类。也就是说，我们只需运行该类的 main() 方法就能启动该应用程序。注意在 main() 方法中必须使用 SpringApplication.run() 方法来运行 HelloServerApplication 类，因为该类启动后将作为一个独立运行在后台的服务，并等待客户端随后发起调用。

默认情况下，该服务启动后将监听本地 8080 端口上的客户端请求，若需修改默认监听的端口号，则可在 application.properties 配置文件中修改如下配置项。

```
server.port=8080
```

下面我们单独创建一个项目，使用 Spring RestTemplate 来调用以上 Spring Boot 开发的服务端应用程序。

4.1.2 使用 Spring RestTemplate 开发客户端

客户端同样也是一个 Spring Boot 应用程序，其 Maven 依赖与服务端相同。

Spring 提供了一个 RestTemplate 类，可调用该类的相关方法来实现 HTTP 调用。在使用时需注意，RestTemplate 不能直接通过依赖注入来获取该实例，必须通过 Spring 所提供的 RestTemplateBuilder 来构建，然而 RestTemplateBuilder 是可以直接依赖注入的。

第一步：创建 RestTemplate 对象。

在 demo.msa.resttemplate.hello.server 包中创建以下 Java 类。

```
package demo.msa.resttemplate.hello.server;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.web.client.RestTemplateBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;

@Configuration
public class RestTemplateConf {

    @Value("${client.root-uri}")
    private String rootUri;

    @Bean
    public RestTemplate restTemplate(RestTemplateBuilder builder) {
        return builder
            .rootUri(rootUri)
            .build();
    }
}
```

我们在 RestTemplateConf 类中使用@Component 注解，定义该类是一个 Spring Bean，可直接使用 Spring 的依赖注入特性。在该类中创建了一个 restTemplate()方法，该方法返回一个 RestTemplate 对象，并传入一个 RestTemplateBuilder 参数（builder），可从 Spring Context 中注入 builder 参数。在 restTemplate()方法中调用 builder 参数的 rootUri()方法来传入服务端 HTTP 请求基础地址，该地址可通过@Value 注解从 application.properties 配置文件中获取，并注入到 rootUri 成员变量中。最后通过调用 builder 参数的 build()方法来创建 RestTemplate 对象。

以下是 application.properties 配置文件的相关代码。

```
server.port=8081

client.root-uri=http://localhost:8080
```

此外，`RestTemplateBuilder` 还提供了一些有价值的功能，比如设置 HTTP 连接超时时间、设置数据读取超时时间等。同样，我们在 `application.properties` 配置文件添加如下配置，可在 `RestTemplateConf` 类中通过 `@Value` 注解将其配置注入进来，最后将其配置传入 `builder` 参数中，从而创建我们所需的 `RestTemplate` 实例。

也可将更多有价值的配置项传入 `RestTemplate` 实例中，比如 HTTP 连接超时时间、数据读取超时时间等，我们在 `application.properties` 配置文件中添加以下配置项。

```
...
```

```
client.connect-timeout=1000
client.read-timeout=1000
```

随后，我们在 `RestTemplateConf` 类中添加以下代码片段。

```
...
```

```
@Configuration
public class RestTemplateConf {

    ...

    @Value("${client.connect-timeout}")
    private int connectTimeout;

    @Value("${client.read-timeout}")
    private int readTimeout;

    @Bean
    public RestTemplate restTemplate(RestTemplateBuilder builder) {
        return builder
            .rootUri(rootUri)
            .setConnectTimeout(connectTimeout) // 设置 HTTP 连接超时时间
            .setReadTimeout(readTimeout) // 设置数据读取超时时间
            .build();
    }
}
```

我们通过调用 `RestTemplateBuilder` 的链式方法来传入相关的配置项，这些配置项都来自

@Value 注解所定义的成员变量中。

既然 RestTemplate 已创建完毕，下面我们将该对象注入到另一个 Spring Bean 中，并通过调用 RestTemplate 的相关方法来发送 HTTP 请求。

第二步：注入并使用 RestTemplate 对象。

在 demo.msa.resttemplate.hello.server 包中创建以下 Java 类。

```
package demo.msa.resttemplate.hello.server;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;

@Component
public class HelloClient {

    @Autowired
    private RestTemplate restTemplate;

    public String say1(String name) {
        return restTemplate.getForObject("/say1?name={name}", String.class, name);
    }

    public String say2(String name) {
        return restTemplate.getForObject("/say2/{name}", String.class, name);
    }
}
```

以上 HelloClient 类同样也需要使用 @Component 注解来定义，因为在类中使用了 @Autowired 注解来注入 RestTemplate 对象。通过调用 RestTemplate 对象的 getForObject() 方法并传入相应的 URL 请求地址、参数类型和参数值，就能发送 GET 类型的 HTTP 请求。

需要注意的是，通过调用 getForObject() 方法只能发送带有请求参数与路径参数的两类 HTTP 请求，如果参数在 HTTP Header 中无法使用以上这种方式来调用，则需要通过调用 RestTemplate 对象的 exchange() 方法将 HTTP Header 参数传递到服务端。

...

```
import org.springframework.beans.factory.annotation.Value;
```

```
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpMethod;
import org.springframework.http.ResponseEntity;
import org.springframework.http.ResponseEntity;

import java.net.URI;

@Component
public class HelloClient {

    ...

    @Value("${client.root-uri}")
    private String rootUri;

    ...

    public String say3(String name) {
        HttpHeaders httpHeaders = new HttpHeaders();
        httpHeaders.add("name", name);
        ResponseEntity<String> requestEntity = new ResponseEntity<>(httpHeaders,
HttpMethod.GET, URI.create(rootUri + "/say3"));
        ResponseEntity<String> responseEntity = restTemplate.exchange
(requestEntity, String.class);
        return responseEntity.getBody();
    }
}
```

我们通过 Spring 提供的 `HttpHeaders` 类来封装 HTTP Header 参数，它本质上是一个 `HashMap` 对象。在调用 `RestTemplate` 对象的 `exchange()` 方法时需要传入一个 `RequestEntity` 对象，创建该对象需要传入 `HttpHeaders` 与具体的请求 URI，此时需要通过 `@Value` 注解从 `application.properties` 配置文件中获取基础 URI 配置。调用 `exchange()` 方法后将返回一个 `ResponseEntity` 对象，最后需从 `ResponseEntity` 对象中获取 HTTP Body 中的数据。

以上我们通过封装 `RestTemplate` 构造了一个名为 `HelloClient` 的 HTTP 客户端类，下面我们将在 Spring Boot 应用启动类中调用该 HTTP 客户端，从而验证我们的服务调用过程。

第三步：发送 HTTP 请求。

在 `demo.msa.resttemplate.hello.server` 包中创建以下 Java 类：

```
package demo.msa.resttemplate.hello.server;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class HelloClientApplication implements CommandLineRunner {

    @Autowired
    private HelloClient helloClient;

    @Override
    public void run(String... args) throws Exception {
        System.out.println(helloClient.say1("world"));
        System.out.println(helloClient.say2("world"));
        System.out.println(helloClient.say3("world"));
    }

    public static void main(String[] args) {
        SpringApplication.run(HelloClientApplication.class, args).close();
    }
}
```

此时我们无须运行一个基于 Web 的 Spring Boot 应用程序作为调用 HTTP 请求的客户端，而是使用 Spring Boot 应用程序启动类 `HelloClientApplication` 实现 `CommandLineRunner` 接口，并在类中重写该接口的 `run()` 方法。接下来在该方法中调用已注入的 `HelloClient` 对象的相关方法，从而以不同方式发送 HTTP 请求并观察调用结果。此时还需注意的是，在 `main()` 方法中调用 `SpringApplication` 类的 `run()` 方法来启动 Spring Boot 应用程序后，我们一定要调用 `close()` 方法来关闭 Spring Boot 应用程序，否则该 Spring Boot 应用程序将在后台持续运行，导致应用程序无法正常退出。

可见，只要我们熟练掌握 Spring 所提供的 `RestTemplate` 的常用 API 使用方法，就能方便地调用 HTTP 请求。但是我们必须提供 `spring-boot-starter-web` 的 Maven 依赖，否则将无法使用

RestTemplate。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

其次，如果我们在本地同时运行服务端和客户端，还需要在 `application.properties` 配置文件中指定客户端的 HTTP 端口，该端口必须与服务端端口不同，否则将无法启动客户端。

```
server.port=8081
```

除了依赖较大，RestTemplate 的 API 使用起来也稍微有些复杂，需要我们投入一定的学习成本。

有没有学习门槛较低，依赖较小的 HTTP 客户端工具呢？我们找到了 OkHttp，它是一款轻量级 HTTP 客户端技术选型。下面我们就尝试使用 OkHttp 来完成以上 RestTemplate 能做到的事情。

4.1.3 使用 OkHttp 开发客户端

OkHttp 是 Square 公司的开源项目之一，它不仅使用方便，而且还支持最新的 HTTP/2 协议，所有发送到同一主机上的请求能共享同一个 Socket 连接，确保在性能方面就得到了很大的提升，此外它也提供了一系列优秀的特性，比如连接池（降低请求延迟）、GZIP（压缩下载大小）、响应缓存（避免重复请求）等。

Square 开源项目：<http://square.github.io/>。

实际上，OkHttp 最早应用于 Android 开发中客户端与服务端的 HTTP 通信，该组件也能用于任何 Java 应用，只是需要提供 1.7 以上版本的 JDK。

我们单独创建一个项目，在 `pom.xml` 配置文件中添加 OkHttp 的 Maven 依赖。

```
<dependency>
  <groupId>com.squareup.okhttp3</groupId>
  <artifactId>okhttp</artifactId>
  <version>3.6.0</version>
</dependency>
```


下面，我们依然分三个步骤来使用 OkHttp 实现 HTTP 请求调用。

第一步：创建 OkHttpClient 对象。

在 demo.msa.okhttp.hello.client 包中创建以下 Java 类。

```
package demo.msa.okhttp.hello.client;

import okhttp3.OkHttpClient;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class OkHttpClientConf {

    @Bean
    public OkHttpClient okHttpClient() {
        return new OkHttpClient();
    }
}
```

这是一个带有@Component 注解的 Spring Bean，其中通过@Bean 注解来标示所创建的 OkHttpClient 对象，如果需要对 OkHttpClient 对象做出相关定制，可以在此处来完成。

第二步：注入并使用 OkHttpClient 对象。

在 demo.msa.okhttp.hello.client 包中创建以下 Java 类：

```
package demo.msa.okhttp.hello.client;

import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.Response;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

import java.io.IOException;

@Component
```

```
public class HelloClient {

    @Autowired
    private OkHttpClient okHttpClient;

    @Value("${client.root-uri}")
    private String rootUri;

    public String say1(String name) throws IOException {
        Request request = new Request.Builder()
            .get()
            .url(String.format("%s/say1?name=%s", rootUri, name))
            .build();
        Response response = okHttpClient.newCall(request).execute();
        return response.body().string();
    }

    public String say2(String name) throws IOException {
        Request request = new Request.Builder()
            .get()
            .url(String.format("%s/say2/%s", rootUri, name))
            .build();
        Response response = okHttpClient.newCall(request).execute();
        return response.body().string();
    }

    public String say3(String name) throws IOException {
        Request request = new Request.Builder()
            .get()
            .url(String.format("%s/say3", rootUri))
            .addHeader("name", name)
            .build();
        Response response = okHttpClient.newCall(request).execute();
        return response.body().string();
    }
}
```

首先需要通过@Autowired 注解将 OkHttpClient 对象注入进来，并通过@Value 注解从

`application.properties` 配置文件中获取基础 URI 地址并注入到 `rootUri` 成员变量中。`say1()` 方法用于发送带请求参数的 HTTP 请求，将请求 URL 构建到 `Request` 对象中，通过 `OkHttpClient` 对象调用 HTTP 请求，将返回的数据通过 `Response` 对象进行封装，最后从 `Response` 对象的 `body` 中获取响应结果。`say2()` 方法用于发送带路径参数的 HTTP 请求，`say3` 方法用于发送参数在请求头中的 HTTP 请求。可见，`OkHttp` 所提供的 API 非常容易使用，而且不同请求方式所使用的 API 风格是一致的，这一点比 Spring 的 `RestTemplate` 要好很多。

第三步：发送 HTTP 请求。

在 `demo.msa.okhttp.hello.client` 包中创建以下 Java 类：

```
package demo.msa.okhttp.hello.client;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class HelloClientApplication implements CommandLineRunner {

    @Autowired
    private HelloClient helloClient;

    @Override
    public void run(String... args) throws Exception {
        System.out.println(helloClient.say1("world"));
        System.out.println(helloClient.say2("world"));
        System.out.println(helloClient.say3("world"));
    }

    public static void main(String[] args) {
        SpringApplication.run(HelloClientApplication.class, args).close();
    }
}
```

通过 `@Autowired` 注解将 `HelloClient` 对象注入其中，通过实现 `CommandLineRunner` 接口的 `run()` 方法来调用 `HelloClient` 对象的相关方法，从而验证 HTTP 请求的调用结果。

此时我们不需要 `spring-boot-starter-web` 的 Maven 依赖，只需依赖 `spring-boot-starter` 即可，

这也是一个很有价值的简化。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter</artifactId>
</dependency>
```

运行 `HelloClientApplication` 应用程序，观察输出是否为我们所期望的结果。

我们可通过 `OkHttp` 官网了解关于它的更多信息，如图 4-1 所示。

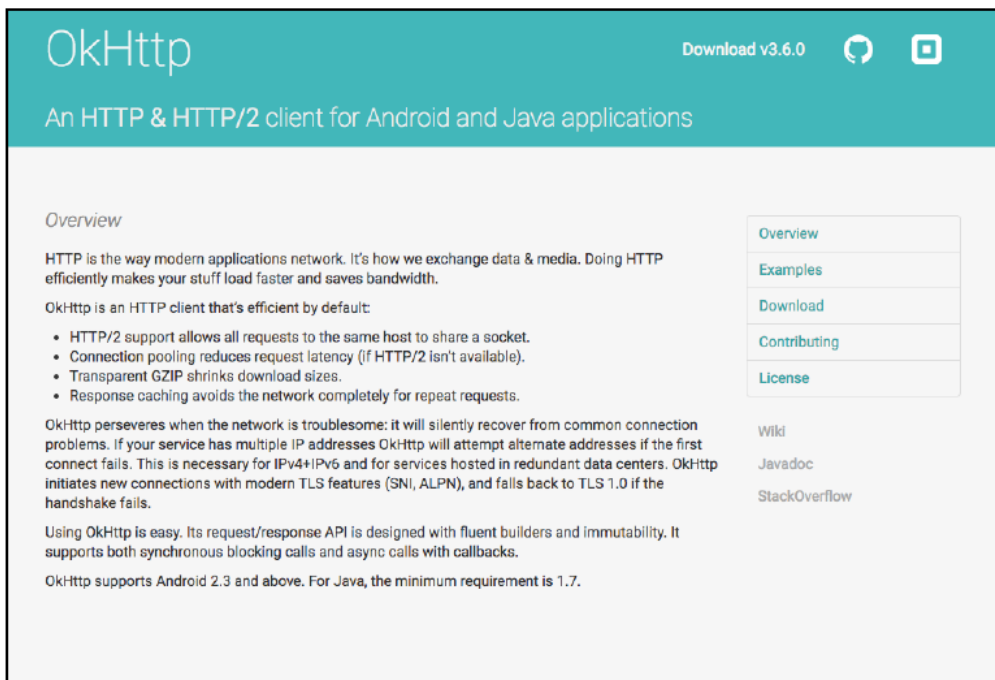


图 4-1 OkHttp 官网

OkHttp 官网：<http://square.github.io/okhttp/>。

除了 OkHttp 开源项目，Square 公司还开源了另一款 HTTP 客户端工具，也非常好用，它的名字叫 Retrofit。下面我们就来学习 Retrofit 的使用方法，看看它是如何实现 HTTP 客户端的。

4.1.4 使用 Retrofit 开发客户端

Retrofit 是 Square 公司基于 OkHttp 开发的一款更加简单易用的 HTTP 客户端工具，我们不

再使用具体的客户端 API，而是根据服务端的 URL 规则来定义所需要调用的客户端接口，在代码中只需调用该客户端接口即可实现 HTTP 调用，无须考虑相应的数据转型问题。这是一种全新的开发方式，它让 HTTP 调用更加自然，下面我们就来体验一下 Retrofit 的强大功能。

首先我们还是需要在 pom.xml 配置文件中添加关于 Retrofit 的 Maven 依赖。

```
<dependency>
  <groupId>com.squareup.retrofit2</groupId>
  <artifactId>retrofit</artifactId>
  <version>2.2.0</version>
</dependency>
```

随后依然根据以下四个步骤，最终完成 HTTP 请求调用。

第一步：创建 Retrofit 对象。

在 demo.msa.retrofit.hello.client 包中创建以下 Java 类：

```
package demo.msa.retrofit.hello.client;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import retrofit2.Retrofit;

@Configuration
public class RetrofitConf {

    @Value("${retrofit.base-url}")
    private String baseUrl;

    @Bean
    public Retrofit retrofit() {
        return new Retrofit.Builder()
            .baseUrl(baseUrl)
            .build();
    }
}
```

通过 Retrofit 提供的 API，我们可以轻松地通过一个链式方法来创建 Retrofit 对象。在创建

Retrofit 对象时需要使用 `@Value` 注解从 `application.properties` 配置文件中获取基础 URL，并将其初始化到 Retrofit 对象的构造过程中。最后不要忘记用 `@Component` 注解将该类定义为一个 Spring Bean 类，它将被 Spring 框架扫描，并调用带有 `@Bean` 注解的方法，从而创建 Retrofit 对象，最终将该对象放入 Spring Context 中。

以下是 `application.properties` 配置文件中需要提供的基础 URL 配置项，它是服务端所暴露的 HTTP 基础地址。

```
retrofit.base-url=http://localhost:8080
```

第二步：定义 HTTP 客户端接口。

在 `demo.msa.retrofit.hello.client` 包中创建以下 Java 类：

```
package demo.msa.retrofit.hello.client;

import retrofit2.Call;
import retrofit2.http.GET;
import retrofit2.http.Header;
import retrofit2.http.Path;
import retrofit2.http.Query;

public interface HelloService {

    @GET("/say1")
    Call<String> say1(@Query("name") String name);

    @GET("/say2/{name}")
    Call<String> say2(@Path("name") String name);

    @GET("/say3")
    Call<String> say3(@Header("name") String name);
}
```

我们创建了一个名为 `HelloService` 的接口，并在该接口中添加了三个方法，分别对应三种类型的 GET 请求。

- 使用 `@Query` 注解定义带请求参数的请求；
- 使用 `@Path` 注解定义带路径参数的请求；

- 使用@Header 注解定义参数在 Request Header 中的请求。

以上三个方法均返回 Call<String>类型的返回值，该返回值带有一个 String 泛型，该泛型表明 HTTP 响应结果中 Response Body 中的数据为 String 类型。

第三步：注入并使用 Retrofit 对象。

在 demo.msa.retrofit.hello.client 包中创建以下 Java 类：

```
package demo.msa.retrofit.hello.client;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import retrofit2.Retrofit;

import java.io.IOException;

@Component
public class HelloClient {

    private HelloService helloService;

    @Autowired
    public HelloClient(Retrofit retrofit) {
        helloService = retrofit.create(HelloService.class);
    }

    public String say1(String name) throws IOException {
        return helloService.say1(name).execute().body();
    }

    public String say2(String name) throws IOException {
        return helloService.say2(name).execute().body();
    }

    public String say3(String name) throws IOException {
        return helloService.say3(name).execute().body();
    }
}
```

我们在 `HelloClient` 类中定义了 `HelloService` 的成员变量，并在该类的构造方法上使用 `@Autowired` 注解将 `Retrofit` 对象注入进来，在构造方法中通过传入 `HelloService` 类型来创建 `HelloService` 对象。随后我们定义了三个方法，分别封装了 `HelloService` 对象的三种调用场景，都是先执行相关操作方法（此时将获取 `Call<String>` 对象），再调用 `execute()` 方法来执行 HTTP 请求操作，最后通过调用 `body()` 方法来获取 `Response Body` 中的数据。由于在调用 `execute()` 方法时会抛出 `IOException` 运行时异常，此时可通过 `try...catch...` 来捕获异常，并进行相关处理；也可向外抛出异常，由调用方来处理，此时我们简单地选择了向外抛出异常。

第四步：发送 HTTP 请求。

在 `demo.msa.retrofit.hello.client` 包中创建以下 Java 类：

```
package demo.msa.retrofit.hello.client;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class HelloClientApplication implements CommandLineRunner {

    @Autowired
    private HelloClient helloClient;

    @Override
    public void run(String... args) throws Exception {
        System.out.println(helloClient.say1("world"));
        System.out.println(helloClient.say2("world"));
        System.out.println(helloClient.say3("world"));
    }

    public static void main(String[] args) {
        SpringApplication.run(HelloClientApplication.class, args).close();
    }
}
```

在以上 `Spring Boot` 应用程序启动类中注入了 `HelloClient` 对象，并在该类的 `run()` 方法中调

用该对象的相关方法，启动 Spring Boot 应用程序，在控制台中观察执行结果。

不幸的是，我们将看到以下异常信息：

```
java.lang.IllegalStateException: Failed to execute CommandLineRunner
    at org.springframework.boot.SpringApplication.callRunner
(SpringApplication.java:779) [spring-boot-1.5.2.RELEASE.jar:1.5.2.RELEASE]
    at org.springframework.boot.SpringApplication.callRunners
(SpringApplication.java:760) [spring-boot-1.5.2.RELEASE.jar:1.5.2.RELEASE]
    at org.springframework.boot.SpringApplication.afterRefresh
(SpringApplication.java:747) [spring-boot-1.5.2.RELEASE.jar:1.5.2.RELEASE]
    at org.springframework.boot.SpringApplication.run(SpringApplication.
java:315) [spring-boot-1.5.2.RELEASE.jar:1.5.2.RELEASE]
    at org.springframework.boot.SpringApplication.run(SpringApplication.
java:1162) [spring-boot-1.5.2.RELEASE.jar:1.5.2.RELEASE]
    at org.springframework.boot.SpringApplication.run(SpringApplication.
java:1151) [spring-boot-1.5.2.RELEASE.jar:1.5.2.RELEASE]
    at demo.msa.retrofit.hello.client.HelloClientApplication.main
(HelloClientApplication.java:25) [classes/:na]
    Caused by: java.lang.IllegalArgumentException: Unable to create converter
for class java.lang.String
    for method HelloService.say1
    at retrofit2.ServiceMethod$Builder.methodError(ServiceMethod.java:751)
~[retrofit-2.2.0.jar:na]
    at retrofit2.ServiceMethod$Builder.createResponseConverter
(ServiceMethod.java:737) ~[retrofit-2.2.0.jar:na]
    at retrofit2.ServiceMethod$Builder.build(ServiceMethod.java:168)
~[retrofit-2.2.0.jar:na]
    at retrofit2.Retrofit.loadServiceMethod(Retrofit.java:169)
~[retrofit-2.2.0.jar:na]
    at retrofit2.Retrofit$1.invoke(Retrofit.java:146)
~[retrofit-2.2.0.jar:na]
    at com.sun.proxy.$Proxy32.say1(Unknown Source) ~[na:na]
    at demo.msa.retrofit.hello.client.HelloClient.say1(HelloClient.
java:22) ~[classes/:na]
    at demo.msa.retrofit.hello.client.HelloClientApplication.run
(HelloClientApplication.java:16) [classes/:na]
    at org.springframework.boot.SpringApplication.callRunner
(SpringApplication.java:776) [spring-boot-1.5.2.RELEASE.jar:1.5.2.RELEASE]
```

```

    ... 6 common frames omitted
    Caused by: java.lang.IllegalArgumentException: Could not locate
    ResponseBody converter for class java.lang.String.
    Tried:
    * retrofit2.BuiltInConverters
    at retrofit2.Retrofit.nextResponseBodyConverter(Retrofit.java:349)
~[retrofit-2.2.0.jar:na]
    at retrofit2.Retrofit.responseBodyConverter(Retrofit.java:311)
~[retrofit-2.2.0.jar:na]
    at retrofit2.ServiceMethod$Builder.createResponseConverter
(ServiceMethod.java:735) ~[retrofit-2.2.0.jar:na]
    ... 13 common frames omitted

```

从异常信息中可知，Retrofit 无法将 `java.lang.String` 类型进行类型转换，因为无法识别 `ResponseBody` 的类型转换器（`Converter`）。也就是说，在默认情况下，Retrofit 无法将 `Call<String>` 返回值中的 `String` 泛型进行转换，我们需要使用 Retrofit 所提供的类型转换器来实现该功能。

在 `pom.xml` 配置文件中，添加以下 Retrofit 类型转换器的 Maven 依赖，它将自动转换 Java 原始类型与包装类型。

```

<dependency>
  <groupId>com.squareup.retrofit2</groupId>
  <artifactId>converter-scalars</artifactId>
  <version>2.2.0</version>
</dependency>

```

除了添加以上 Maven 依赖，我们还需将对应的类型转换器添加到 Retrofit 对象的构造过程中。

```

...

import retrofit2.converter.scalars.ScalarsConverterFactory;

@Configuration
public class RetrofitConf {

    @Value("${retrofit.base-url}")
    private String baseUrl;

```

```

@Bean
public Retrofit retrofit() {
    return new Retrofit.Builder()
        .baseUrl(baseUrl)
        // 自动转换 Java 原始类型与包装类型
        .addConverterFactory(ScalarsConverterFactory.create())
        .build();
}
}

```

再次运行 Spring Boot 应用程序，将返回我们想要的结果。

既然 String 这样的 Java 包装类型可以自动转换，那么对于集合类型，比如 List、Map 能否也自动转换呢？此外，对于普通的 Bean 类型，是否同样也能自动转换呢？

对于 List、Map、Bean 等 Java 类型而言，SpringBoot 服务端自动将它们序列化为一个 JSON 字符串，此时我们需要将此 JSON 字符串自动反序列化为对应的 Java 类型。

此时，我们需要在 pom.xml 配置文件中添加 Retrofit 另一个自带的 Maven 依赖，它提供了一个基于 Jackson 开源包的 JSON 数据转换器。

```

<dependency>
    <groupId>com.squareup.retrofit2</groupId>
    <artifactId>converter-jackson</artifactId>
    <version>2.2.0</version>
</dependency>

```

同样需要在 Retrofit 对象的构造过程中添加以下类型转换器。

```

...

import retrofit2.converter.jackson.JacksonConverterFactory;

@Configuration
public class RetrofitConf {

    @Value("${retrofit.base-url}")
    private String baseUrl;

    @Bean

```

```
public Retrofit retrofit() {  
    return new Retrofit.Builder()  
        .baseUrl(baseUrl)  
        .addConverterFactory(ScalarsConverterFactory.create())  
        // 自动转换 JSON 字符串类型  
        .addConverterFactory(JacksonConverterFactory.create())  
        .build();  
}  
}
```

请大家自行尝试在服务端添加 List、Map、Bean 三种类型的 HTTP 接口，并通过扩展 Retrofit 的客户端 HTTP 接口来验证以上 JSON 数据的自动转换过程。

我们可通过 Retrofit 官网了解关于它的更多信息，如图 4-2 所示。

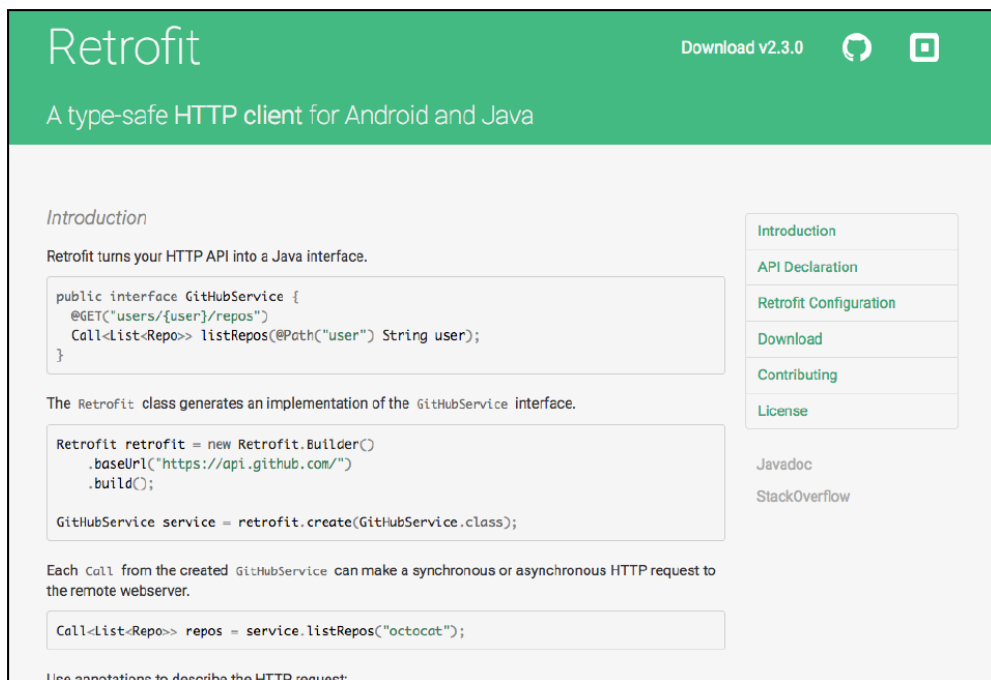


图 4-2 Retrofit 官网

Retrofit 官网：<http://square.github.io/retrofit/>。

以上我们使用 Spring Boot 开发了服务端，并分别体验了三种 HTTP 客户端工具。对于 Spring 自带的 RestTemplate 而言，它的使用方法较为简单，只是 API 调用方式不太一致，此外所需的

依赖也相对较多。OkHttp 所提供的链式调用方法使我们发送 HTTP 请求变得相当方便，依赖也变得相对较少。

Retrofit 是一种全新的开发方式，我们只需根据目标 HTTP 地址，在客户端中定义相关的服务接口，并通过注解的方式来绑定 HTTP 请求。以上三种 HTTP 客户端工具各有千秋，我们可根据自身需求灵活地选择最佳的解决方案。

4.2 使用 RPC 实现同步调用

在上一节中，我们已经探讨了基于 HTTP 的同步调用方式，它能很方便地解决服务之间的调用问题。但是，当服务之间的调用较为频繁时，我们一般不会选择 HTTP 调用，而是选择基于 TCP 的 RPC 调用。所谓 RPC 即 Remote Procedure Call（远程过程调用），这种方式可确保调用性能更加高效，能支持更高的并发量。因为 RPC 通信过程在传输层中完成（HTTP 通信过程在应用层中完成），所以使用 RPC 调用方式需要服务端与客户端之间建立 Socket 连接来实现二进制数据的交换。RPC 底层调用过程非常复杂，我们一般会使用 RPC 框架将这些复杂的细节加以封装。业界流行的 RPC 框架都比较好用，比如 Google gRPC、Facebook Thrift、Twitter Finagle、阿里巴巴 Dubbo、新浪微博 Motan 等，使用这些 RPC 框架可显著提高我们的开发效率，降低我们的使用门槛，让我们将更多的精力从底层细节中释放出来，去关注上层的业务需求实现。

在本节中，我们首先会对 RPC 通信原理进行简单描述，确保大家对 RPC 调用过程有一个清晰的认识，随后将以 Google gRPC 框架为例，学习 RPC 框架的使用方法，最后在 gRPC 框架的基础上进行简单地封装，将其集成到 Spring Boot 框架中。

4.2.1 RPC 通信原理

假设有两个应用程序，它们分别运行在不同的机器上，我们需要使用其中一个应用程序来调用另一个应用程序。为了便于描述，我们将调用方应用程序称为 Client，将被调用方应用程序称为 Server，图 4-3 所示便是 Client 调用 Server 的整个 RPC 通信过程。

- (1) Client 应用程序调用 Client Stub（客户端存根），在 Client Stub 中对请求数据进行序列化操作。
- (2) 在 Client Stub 中创建本地 Socket 连接。
- (3) 客户端机器与服务端机器在网络上建立 Socket 连接。
- (4) 在服务端调用 Server Stub（服务端存根），在 Server Stub 中对数据进行反序列化操作。
- (5) Server Stub 调用 Server 应用程序，在 Server 应用程序中处理业务逻辑。

- (6) Server 应用程序调用 Server Stub，在 Server Stub 中对响应数据进行序列化操作。
- (7) 在 Server Stub 中创建本地 Socket 连接。
- (8) 服务端机器与客户端机器在网络上建立 Socket 连接。
- (9) 在客户端中调用 Client Stub，在 Client Stub 中对数据进行反序列化操作。
- (10) Client Stub 调用 Client 应用程序，在 Client 应用程序中获取响应结果。

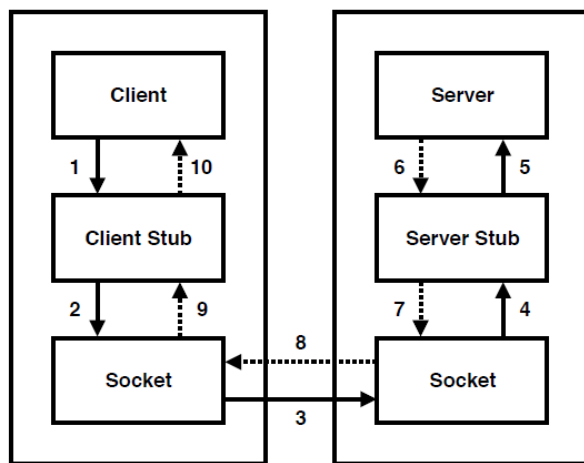


图 4-3 RPC 通信过程

可见，一次 RPC 调用过程涉及许多底层的通信细节，以上描述只是 RPC 调用的核心步骤，实际情况远比这个过程要复杂许多。需要在 Stub 中对数据进行序列化与反序列化，也需要在远程网络之间建立 Socket 连接，只要将这些复杂的技术环节封装在 RPC 框架中，我们就能将精力集中在业务上。通过 RPC 框架来屏蔽底层的通信过程，这是 RPC 框架的核心价值。

下面我们就以 Google 开源的 RPC 框架 gRPC 为例，体验一下它给我们工作上带来的便捷，还有它的价值。

4.2.2 初步体验 gRPC

gRPC 是 Google 开源的高性能通用 RPC 框架，它可适用于众多开发语言中，包括 C++、Java、Python、Go、Ruby、C#、Node.js、Android、Objective-C、PHP 等。它使用 Protocol Buffers 来编写服务定义，拥有强大的二进制序列化工具集，同样也支持多种开发语言。它能自动生成客户端与服务端 Stub，并支持跨语言与跨平台的特性。我们只需使用一行命令就能快速搭建 RPC 运行环境，并支持高并发量的 RPC 调用。此外，客户端与服务端还能双向支持基于 HTTP 的安全认证插件。

gRPC 与 Protocol Buffers 都是 Google 推出的开源项目，我们可访问它们的官网来了解更多相关信息，如图 4-4 与图 4-5 所示。

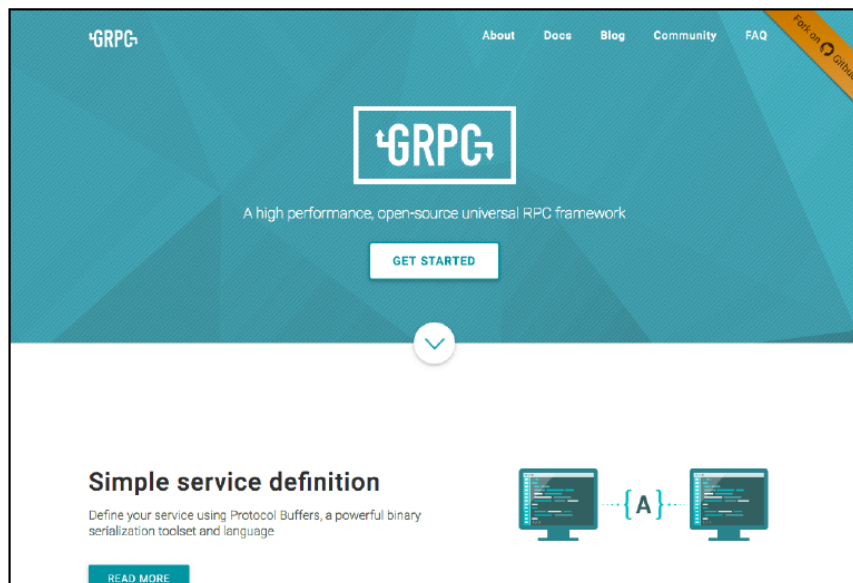


图 4-4 gRPC 官网

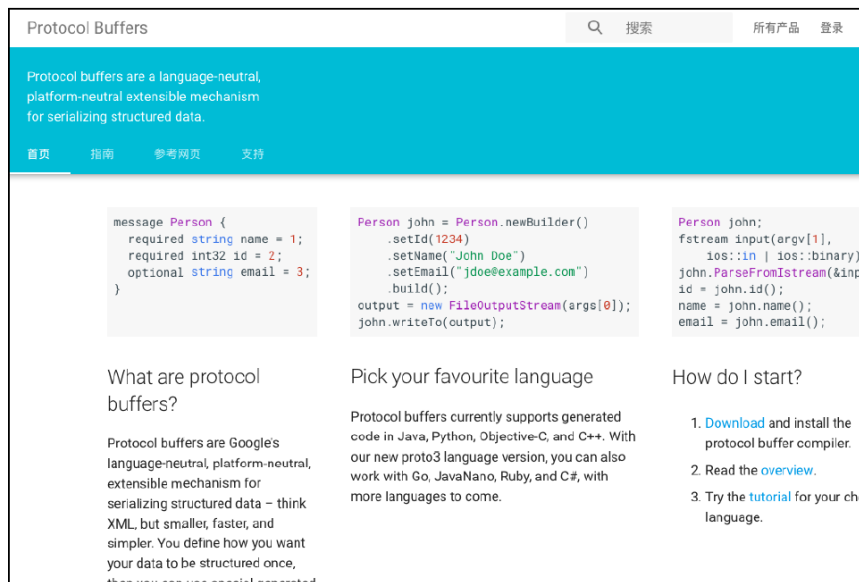


图 4-5 Protocol Buffers 官网

gRPC 官网：<http://www.grpc.io/>。

Protocol Buffers 官网：<https://developers.google.com/protocol-buffers/>。

下面我们通过三个步骤来体验 gRPC 的开发过程。

第一步：定义 RPC 接口。

创建一个名为 `grpc-hello-api` 的 Maven 项目，在 `pom.xml` 配置文件中添加 gRPC 的 Maven 依赖。

```
<dependencies>
  <dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-all</artifactId>
    <version>1.2.0</version>
  </dependency>
</dependencies>
```

`grpc-all` 依赖包含了 gRPC 所拥有的全部依赖，简单情况下我们可直接使用这个依赖，特定情况下也能有针对性地选择 gRPC 的依赖，比如 `grpc-netty`、`grpc-protobuf`、`grpc-stub` 等。

此外，我们还需要添加以下 Maven 扩展与插件，因为后面需要直接使用 Maven 来执行 Protocol Buffers 命令，从而生成相关的 Stub 代码库。

```
<build>
  <extensions>
    <extension>
      <groupId>kr.motd.maven</groupId>
      <artifactId>os-maven-plugin</artifactId>
      <version>1.4.1.Final</version>
    </extension>
  </extensions>
  <plugins>
    <plugin>
      <groupId>org.xolstice.maven.plugins</groupId>
      <artifactId>protobuf-maven-plugin</artifactId>
      <version>0.5.0</version>
      <configuration>
        <pluginId>grpc-java</pluginId>
        <protocArtifact>com.google.protobuf:protoc:3.0.2:exe:${os.detect
```



```

ed.classifier}</protocArtifact>
    <pluginArtifact>io.grpc:protoc-gen-grpc-java:1.2.0:exe:${os.detected.classifier}</pluginArtifact>
</configuration>
<executions>
    <execution>
        <goals>
            <goal>compile</goal>
            <goal>compile-custom</goal>
        </goals>
    </execution>
</executions>
</plugin>
</plugins>
</build>

```

os-maven-plugin 扩展用于生成各种平台无关的属性，protobuf-maven-plugin 插件用于执行 Protocol Buffers 命令并生成 Stub 代码库。从以上配置中可知，当我们执行 mvn compile 命令时将生成 Stub 代码库。

随后，我们在 src/main 目录下创建一个名为 proto 的目录，并在该目录下创建一个名为 hello.proto 的文件，其文件内容如下所示。

```

// 定义语法版本
syntax = "proto3";

// 定义 Stub 代码选项
option java_package = "demo.msa.grpc.hello.api";
option java_outer_classname = "HelloProto";
option java_multiple_files = true;

// 定义包名
package demo.msa.grpc.hello.api;

// 定义服务
service HelloService {
    // 定义方法
    rpc Say (HelloRequest) returns (HelloResponse) {}
}

```

```
// 定义消息（请求）
message HelloRequest {
    string name = 1;
}

// 定义消息（响应）
message HelloResponse {
    string message = 1;
}
```

hello.proto 是基于 Protocol Buffers 规范编写的服务定义，使用 syntax 定义 Protocol Buffers 文件的语法版本，使用 option 定义所生成 Java 源代码（即 Stub 代码库）的相关选项。

- `java_package`: 生成 Stub 代码所对应的包名。
- `java_outer_classname`: 生成 Stub 代码外部类名。
- `java_multiple_files`: 生成 Stub 代码是否为多份文件。

使用 `package` 指定服务接口的包名，使用 `service` 定义服务，在服务中使用 `rpc` 定义方法，在一个服务中可定义一个或多个方法，方法包括方法名、参数与返回值。此处的方法名是 `Say`，参数是 `HelloRequest` 消息，返回值是 `HelloResponse` 消息。所有的消息均使用 `message` 来定义，消息包括一个消息名、一个或多个相关属性，属性包括属性类型、属性名称、属性顺序。以 `HelloRequest` 消息为例，它拥有一个属性，该属性类型为 `string`，属性名称为 `name`，该属性出现在第 1 位。

现在可执行 `mvn compile` 命令，观察 `target/generated-sources` 目录下自动生成的 Stub 代码库，如图 4-6 所示。

既然 Stub 代码库已经生成，那么下一步就可以使用这个 Stub 代码库开发服务端与客户端了，我们不妨先从服务端开始。

第二步：开发 RPC 服务端。

创建一个名为 `grpc-hello-server` 的 Maven 项目，在 `pom.xml` 配置文件中添加 `grpc-hello-api` 的 Maven 依赖。

```
<dependencies>
```

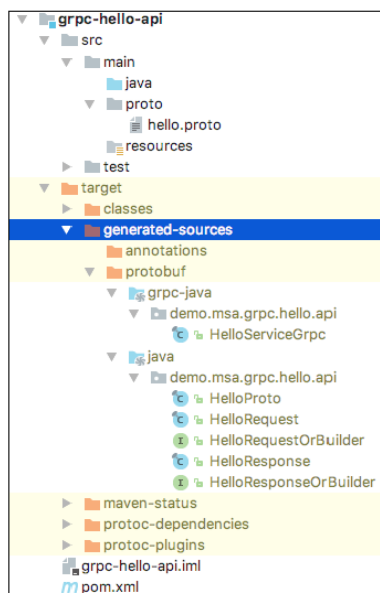


图 4-6 gRPC 生成的 Stub 代码库

```

<dependency>
  <groupId>demo.msa</groupId>
  <artifactId>grpc-hello-api</artifactId>
  <version>1.0.0</version>
</dependency>
</dependencies>

```

通过以上 Maven 配置，我们可在 `grpc-hello-server` 项目中直接使用 `grpc-hello-api` 项目所生成的 Stub 代码库。由于 Maven 依赖有传递性，那么我们也能在 `grpc-hello-server` 项目中使用 `grpc-all` 依赖中所包含的代码。

我们在 `grpc-hello-api` 项目中编写的 `hello.proto` 文件是一份服务定义描述，通过调用 Maven 来执行 gRPC 命令，从而生成 Stub 代码库，在这份 Stub 代码库中拥有一套 API 接口，我们可在服务端中实现该接口，从而完成接口的相关实现。该接口位于 `grpc-hello-api` 项目的 `demo.msa.grpc.hello.api` 包中，其中有个名为 `HelloServiceGrpc` 的类，在该类中有个名为 `HelloServiceImplBase` 的抽象类，我们只需在服务端中继承该抽象类，并重写抽象类中的 `say()` 方法，就能完成 API 接口的业务逻辑实现。

以下 `HelloServiceImpl` 类是 API 接口的实现类，我们不仅使用了 `grpc-hello-api` 项目生成的 Stub 代码库，还使用了 gRPC 核心成员 `StreamObserver` 来完成服务端响应过程。

```

package demo.msa.grpc.hello.server;

import demo.msa.grpc.hello.api>HelloRequest;
import demo.msa.grpc.hello.api>HelloResponse;
import demo.msa.grpc.hello.api>HelloServiceGrpc;
import io.grpc.stub.StreamObserver;

public class HelloServiceImpl extends HelloServiceGrpc>HelloServiceImplBase {

    @Override
    public void say>HelloRequest request, StreamObserver<HelloResponse>
responseObserver) {
        HelloResponse response = null;
        try {
            response = HelloResponse
                .newBuilder()
                .setMessage("hello " + request.getName())
                .build();

```

```
        } catch (Exception e) {
            responseObserver.onError(e);
        } finally {
            responseObserver.onNext(response);
        }
        responseObserver.onCompleted();
    }
}
```

我们在 `HelloServiceImpl` 类中继承了 `HelloServiceGrpc>HelloServiceImplBase` 抽象类，并重写了抽象类中的 `say()` 方法，该方法中带有两个参数，可从 `HelloRequest` 参数中获取请求参数，可通过 `StreamObserver<HelloResponse>` 参数完成响应过程。我们首先定义了一个空的 `HelloResponse` 对象，随后通过一个 `try...catch...finally...` 结构来执行响应过程。在 `try` 中通过 `HelloResponse` 的链式方法来构建响应对象；在 `catch` 中捕获异常信息，并将该异常传入 `StreamObserver<HelloResponse>` 参数中；在 `finally` 中将 `HelloResponse` 对象添加到 `StreamObserver<HelloResponse>` 参数中。在 `say()` 方法的最后一行结束整个响应过程。

随后，我们需要创建一个服务端应用程序启动类，它将在指定的端口号上启动 RPC 服务。

以下 `HelloServer` 类是服务端应用程序启动类，我们使用了 gRPC 核心成员 `Server` 与 `ServerBuilder` 来完成服务端启动过程。

```
package demo.msa.grpc.hello.server;

import io.grpc.Server;
import io.grpc.ServerBuilder;

public class HelloServer {

    private static final int port = 8000;

    public static void main(String[] args) throws Exception {
        Server server = ServerBuilder
            .forPort(port)
            .addService(new HelloServiceImpl())
            .build()
            .start();

        System.out.println("server started, listening on " + port);
        server.awaitTermination();
    }
}
```

```
    }
}
```

我们在 `HelloServer` 类中定义了一个端口号常量，在 `main()` 方法中使用 `ServerBuilder` 类来构建 `Server` 对象，此时需要调用 `forPort()` 方法来设置 RPC 端口号，并调用 `addService()` 方法来添加服务实现类实例对象，随后调用 `build()` 方法来构建 `Server` 对象，最后在 `Server` 对象上调用 `start()` 方法来启动 RPC 服务器。当 RPC 服务器启动成功后可在控制台中打印所监听的端口号，并调用 `Server` 对象的 `awaitTermination()` 方法来等待 RPC 服务器终止，从而停止服务端应用程序。

现在我们立即运行 `HelloServer` 应用程序，如果控制台输出了 RPC 端口号，那么就说明 RPC 服务端启动正常，随后我们可通过 RPC 客户端来调用 RPC 服务端在此端口号上暴露的 RPC 接口。

第三步：开发 RPC 客户端。

创建一个名为 `grpc-hello-client` 的 Maven 项目，在 `pom.xml` 配置文件中添加 `grpc-hello-api` 的 Maven 依赖。

```
<dependencies>
  <dependency>
    <groupId>demo.msa</groupId>
    <artifactId>grpc-hello-api</artifactId>
    <version>1.0.0</version>
  </dependency>
</dependencies>
```

以上过程与 RPC 服务端完全相同，也就是说，不管是 RPC 服务端还是 RPC 客户端，它们都依赖于 RPC 接口。RPC 服务端用于实现 RPC 接口，并在指定的端口号上启动 RPC 服务器。RPC 客户端通过 RPC 接口和相应的端口号来调用 RPC 服务端中的 RPC 接口实现。

以下 `HelloClient` 类就是 RPC 客户端的所有代码，我们不仅使用了 `grpc-hello-api` 项目生成的 Stub 代码库，还使用了 gRPC 核心成员 `ManagedChannel` 与 `ManagedChannelBuilder` 来完成客户端请求过程。

```
package demo.msa.grpc.hello.client;

import demo.msa.grpc.hello.api.HelloRequest;
import demo.msa.grpc.hello.api.HelloResponse;
import demo.msa.grpc.hello.api.HelloServiceGrpc;
import io.grpc.ManagedChannel;
```

```
import io.grpc.ManagedChannelBuilder;

public class HelloClient {

    private static final String host = "localhost";
    private static final int port = 8000;

    public static void main(String[] args) throws Exception {
        ManagedChannel channel = ManagedChannelBuilder
            .forAddress(host, port)
            .usePlaintext(true)
            .build();
        try {
            HelloServiceGrpc.HelloServiceBlockingStub helloService =
HelloServiceGrpc.newBlockingStub(channel);
            HelloRequest request = HelloRequest
                .newBuilder()
                .setName("world")
                .build();
            HelloResponse response = helloService.say(request);
            System.out.println(response.getMessage());
        } finally {
            channel.shutdown();
        }
    }
}
```

我们在 `HelloClient` 类中定义了 RPC 服务端所在的主机名与端口号所对应的常量，在 `main()` 方法中通过 `ManagedChannelBuilder` 类来构建 `ManagedChannel` 对象，此时需要调用 `forAddress()` 方法来传入主机名与端口号，并调用 `usePlaintext()` 方法来使用纯文本方式进行数据传输，最后调用 `build()` 方法来构建 `ManagedChannel` 对象。随后我们使用 `grpc-hello-api` 项目中生成的 `HelloServiceGrpc` 类来创建 `Stub` 对象（`helloService` 对象），并通过 `HelloRequest` 类来构建请求对象，此时需要传递请求参数到该对象中，才能调用 `Stub` 对象的目标方法（`say()` 方法）来执行 RPC 调用，并打印出 RPC 响应结果。最后调用 `ManagedChannel` 对象的 `shutdown()` 方法来停止 RPC 客户端应用程序。

需要注意的是，此时创建的 `Stub` 对象是阻塞式的（`Blocking`），这也意味着此时的调用过程是同步式的，但这并不意味着 gRPC 只支持同步式 RPC 调用，其实 gRPC 也支持异步式 RPC 调

用，现在我们只讨论同步式 PRC。

运行 HelloClient，观察控制台中是否出现 hello world 输出，从而验证 RPC 调用是否成功。

以上便是 gRPC 调用的整个过程，该过程分三个步骤来完成。首先我们需要使用 Protocol Buffers 规范对 RPC 接口进行定义，从而生成服务端与客户端所需的 Stub 代码库；随后我们开发了 RPC 服务端，通过继承 Stub 代码库中所提供的抽象类来完成服务端业务逻辑实现，并在特定的端口号上启动 RPC 服务器；最后我们仍然使用 Stub 代码库来编写 RPC 客户端，需要使用 Stub 代码库来构建 Stub 对象与 RPC 请求对象，并通过 Stub 对象来发送 RPC 请求，最终获取 RPC 响应对象。

对于 RPC 服务端而言，需要创建 RPC 接口实现类实例对象，并将其添加到 gRPC 的 ServerBuilder 中，如果服务端中存在大量的 RPC 接口实现类，那么我们需要一个个手工将其添加到 ServerBuilder 中。能否在 RPC 接口实现类中添加一个注解，当服务端启动时自动扫描带有该注解的类，并自动将其添加到 ServerBuilder 中呢？此外，对于 RPC 服务端与客户端而言，主机名与端口号完全可以从配置文件中获取，那么从配置中创建 Server 与 ManagedChannel 这个过程能否在 Spring Boot 框架中完成呢？

带着以上这些问题，我们不妨尝试将 Spring Boot 与 gRPC 进行集成，让 Spring Boot 应用程序具备 RPC 通信的特性。

4.2.3 Spring Boot 集成 gRPC

对于 RPC 服务端而言，我们需要开发一个名为 @GrpcService 的注解，将该注解添加到 RPC 接口实现类上，在 Spring Boot 应用程序启动时加载该注解所对应的类实例，并将这些类实例添加到 ServerBuilder 中。

创建一个名为 grpc-server 的 Maven 项目，该项目中拥有 Spring Boot 与 gRPC 所需的 Maven 依赖。

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot</artifactId>
    <version>1.5.2.RELEASE</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>io.grpc</groupId>
```

```
<artifactId>grpc-all</artifactId>
<version>1.2.0</version>
<scope>provided</scope>
</dependency>
</dependencies>
```

在 `grpc-server` 项目中，创建一个名为 `@GrpcService` 的注解类。

```
package demo.msa.grpc.server;

import org.springframework.stereotype.Service;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Service
public @interface GrpcService {
}
```

该注解没有任何属性，仅为标示性注解，可将其用于 RPC 接口实现类上。但该注解拥有 Spring 的 `@Service` 注解的特性，因此 Spring 框架可自动扫描带有 `@GrpcService` 注解的类，通过反射的方式为其创建类实例，并将该实例作为 Spring Bean，将其放入 Spring 的 `ApplicationContext` 中，从而为 Spring 的 IoC 框架提供基础支持。

```
package demo.msa.grpc.hello.server;

import demo.msa.grpc.server.GrpcService;

@GrpcService
public class HelloServiceImpl extends HelloServiceGrpc.HelloServiceImplBase {
    ...
}
```

在 `grpc-server` 项目中，创建一个名为 `GrpcServer` 的类，其将实现 Spring 的 `ApplicationContextAware`

与 `InitializingBean` 接口。

```
package demo.msa.grpc.server;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.stereotype.Component;

@Component
public class GrpcServer implements ApplicationContextAware, InitializingBean {

    @Override
    public void setApplicationContext(ApplicationContext applicationContext)
throws BeansException {
    }

    @Override
    public void afterPropertiesSet() throws Exception {
    }
}
```

其中，`setApplicationContext()`方法来自 `ApplicationContextAware` 接口，可在该方法中获取 Spring 的 `ApplicationContext` 对象，从而可获取带有 `@GrpcService` 注解的 RPC 接口实现类实例；`afterPropertiesSet()`方法来自 `InitializingBean` 接口，可在该方法中绑定 RPC 端口号，并添加所有 RPC 接口实现类实例，最后启动 RPC 服务器。

由于添加到 `ServerBuilder` 对象中的 RPC 接口实现类实例是 `BindableService` 类型的，因此我们需要定义一个 `List<BindableService>` 类型的成员变量，在 `setApplicationContext()`方法中初始化 `List<BindableService>` 对象，并在 `afterPropertiesSet()`方法中获取 `List<BindableService>` 对象。

以下是 `GrpcServer` 的所有代码，可清晰地看到在 `setApplicationContext()`与 `afterPropertiesSet()`方法中分别完成了各自负责的任务。

```
package demo.msa.grpc.server;

import io.grpc.BindableService;
import io.grpc.Server;
import io.grpc.ServerBuilder;
```

```
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.stereotype.Component;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;

@Component
public class GrpcServer implements ApplicationContextAware, InitializingBean {

    private List<BindableService> serviceBeanList = new ArrayList<>();

    @Value("${grpc.port}")
    private int port;

    @Override
    public void setApplicationContext(ApplicationContext applicationContext)
throws BeansException {
        Map<String, Object> serviceBeanMap =
applicationContext.getBeansWithAnnotation (GrpcService.class);
        if (serviceBeanMap != null && serviceBeanMap.size() != 0) {
            for (Object serviceBean : serviceBeanMap.values()) {
                if (serviceBean instanceof BindableService) {
                    serviceBeanList.add((BindableService) serviceBean);
                }
            }
        }
    }

    @Override
    public void afterPropertiesSet() throws Exception {
        ServerBuilder builder = ServerBuilder.forPort(port);
        for (BindableService serviceBean : serviceBeanList) {
            builder.addService(serviceBean);
        }
    }
}
```

```

    }
    Server server = builder.build().start();
    System.out.println("server started, listening on " + port);
    server.awaitTermination();
}
}

```

需要补充说明的是，在 `GrpcServer` 类中，我们通过 `@Value` 注解来获取 `application.properties` 配置文件中的 `grpc.port` 配置项，它是 RPC 服务端需要暴露的端口号。

```
grpc.port=8000
```

通过以上准备，我们可直接启动 `Spring Boot` 应用程序，框架将自动扫描 `@GrpcService` 注解，并启动 RPC 服务器。

```

package demo.msa.grpc.hello.server;

import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication(scanBasePackages = "demo.msa.grpc")
public class HelloServerApplication implements CommandLineRunner {

    @Override
    public void run(String... args) throws Exception {
    }

    public static void main(String[] args) {
        SpringApplication.run(HelloServerApplication.class, args);
    }
}

```

需要注意的是，我们一定要在 `@SpringBootApplication` 注解中指定扫描包的基础路径（`demo.msa.grpc`），否则只能扫描 `HelloServerApplication` 所在的包（`demo.msa.grpc.hello.server`），从而导致无法加载 `GrpcServer` 类，也就无法成功启动 RPC 服务器了。

对于 RPC 客户端而言，我们也有必要将主机名与端口号进行封装，并提供一个能够获取 `ManagedChannel` 对象的方法，随后可通过 `ManagedChannel` 对象来创建 `Stub` 对象并发送 RPC

请求。

以下创建了一个 `GrpcClient` 类，用于封装 RPC 客户端代码，同样通过 `@Value` 注解从 `application.properties` 配置文件中获取 `grpc.host` 主机名与 `grpc.port` 端口号。

```
package demo.msa.grpc.client;

import io.grpc.ManagedChannel;
import io.grpc.ManagedChannelBuilder;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class GrpcClient {

    @Value("${grpc.host}")
    private String host;

    @Value("${grpc.port}")
    private int port;

    public ManagedChannel buildChannel() {
        return ManagedChannelBuilder
            .forAddress(host, port)
            .usePlaintext(true)
            .build();
    }
}
```

接下来，我们只需将 `GrpcClient` 对象注入到 Spring Boot 应用程序启动类中，从而完成后续的 RPC 调用。

```
package demo.msa.grpc.hello.client;

import demo.msa.grpc.client.GrpcClient;
import demo.msa.grpc.hello.api.HelloRequest;
import demo.msa.grpc.hello.api.HelloResponse;
import demo.msa.grpc.hello.api.HelloServiceGrpc;
import io.grpc.ManagedChannel;
```

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication(scanBasePackages = "demo.msa.grpc")
public class HelloClientApplication implements CommandLineRunner {

    @Autowired
    private GrpcClient grpcClient;

    @Override
    public void run(String... args) throws Exception {
        ManagedChannel channel = grpcClient.buildChannel();
        try {
            HelloServiceGrpc.HelloServiceBlockingStub helloService =
HelloServiceGrpc.newBlockingStub(channel);
            HelloRequest request = HelloRequest
                .newBuilder()
                .setName("world")
                .build();
            HelloResponse response = helloService.say(request);
            System.out.println(response.getMessage());
        } finally {
            channel.shutdown();
        }
    }

    public static void main(String[] args) {
        SpringApplication.run(HelloClientApplication.class, args).close();
    }
}

```

在以上 Spring Boot 应用程序启动类中，同样需要指定扫描包的基础路径，否则无法加载 GrpcClient 实例。

通过以上对 gRPC 的简单封装，我们将 gRPC 轻松地集成到 Spring Boot 框架中，但在应用程序中仍然无法摆脱对 gRPC 代码层面上的依赖。比如在 HelloServiceImpl 类中，我们仍然需要

使用 gRPC 的 API 与生成的 Stub 代码库来实现业务逻辑。再比如在 HelloClientApplication 类中，我们也需要使用 gRPC 的 API 与生成的 Stub 代码库来发送 RPC 请求。也就是说，在 Spring Boot 应用中，我们对 gRPC 仍然存在代码层面上的耦合，这是目前所面临的问题，我们必须对 gRPC 进行更深层次的封装，才能从根本上解决这个问题。

我们了解到 gRPC 底层使用了 Netty 来启动 RPC 服务器，并通过 Netty 来实现 RPC 通信，满足了高性能的要求。此外，gRPC 底层使用了 Protocol Buffers 来实现对象序列化与反序列化操作，做到了跨平台的特性。我们能否也使用 Netty 与 Protocol Buffers，搭建一套能够与 Spring Boot 更加无缝整合的分布式 RPC 框架呢？带着这个问题，我们继续探险吧。

4.3 搭建分布式 RPC 框架

为了解决分布式应用程序之间的 RPC 调用问题，我们决定自行开发一套 RPC 框架。这款 RPC 框架需要做到足够轻量级，首先功能必须够用且用法必须简单，其次性能必须高且稳定性必须有保障。我们会将 Netty 作为主要的技术选型，同时也会用到一些其他相关的开源技术。在这一节中，我们会从零开始，搭建这款轻量级分布式 RPC 框架，它所提供的功能也许看起来可能比较单薄，甚至在某些细节上可能缺乏更加细致的考虑，但这些绝不会影响我们对轻量级系统架构的追求。我们只需根据自己的实际工作情况，在此框架上稍加改进，就能开发出一款更加易用、更加灵活、更加强大的分布式 RPC 框架。

相信大家已经迫不及待了，精彩的架构探险过程即将开启，我们先从“架构设计”这一站出发。

4.3.1 架构设计

对于这款分布式 RPC 框架而言，我们希望它具备以下基本特性。

- 具备高性能、高并发、高可用能力。
- 客户端无须知道服务端的具体信息。
- 客户端与服务端均可面向接口编程。

为了满足第一点特性，我们可通过 Netty 框架来实现高性能与高并发，因为 Netty 提供了 NIO（Non-blocking I/O，非阻塞输入/输出）支持，具备异步通信与事件驱动特性，可用于开发 RPC 框架的服务端与客户端。

关于 Netty 的更多信息，可通过它的官网进行学习，如图 4-7 所示。

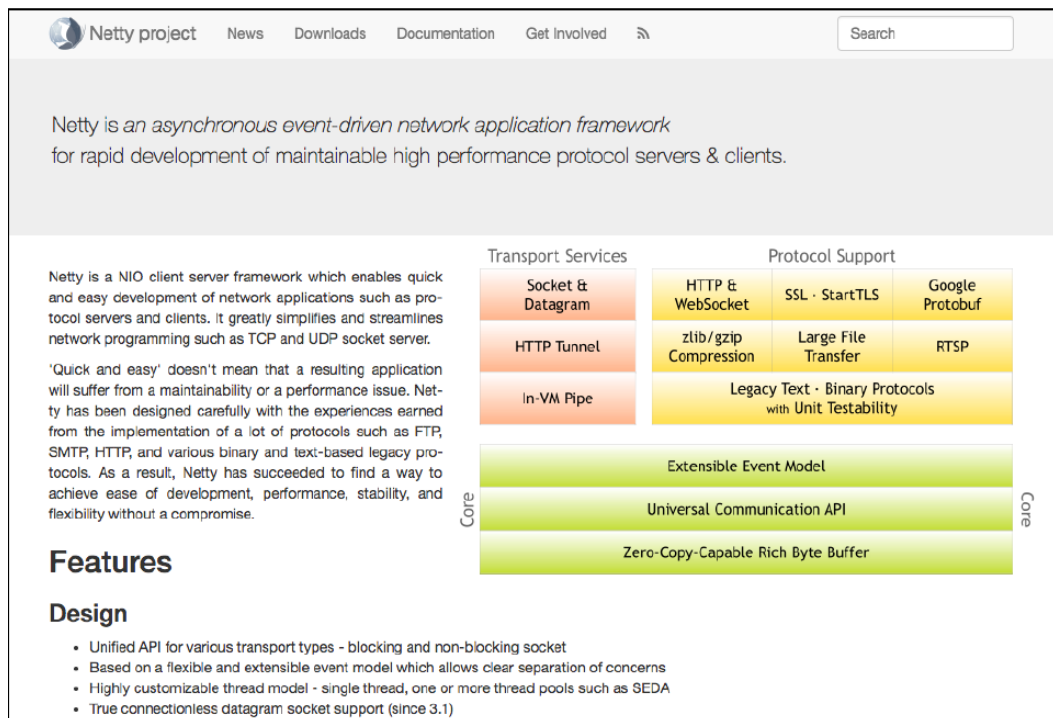


图 4-7 Netty 官网

Netty 官网: <http://netty.io/>。

即便我们使用了 Netty, 但是也无法确保 RPC 框架的高可用特性, 如果服务端崩溃了, 此时将直接影响客户端的正常调用。如何解决该问题呢?

我们不妨使用 ZooKeeper 来注册服务端的配置信息 (包括服务名称、主机名、端口号等), 在客户端通过 RPC 来调用服务端之前, 需要先从 ZooKeeper 这个“服务注册中心”中获取服务端的配置信息 (该操作称为“服务发现”), 进而通过主机名与端口号去连接相应的服务端, 随后通过调用本地接口来执行 RPC 请求的远程调用, 最终获取服务端的响应结果。更重要的是, ZooKeeper 与服务端建立了长连接, 可定时进行“心跳检测”, 确保服务端是否正常运行, 当进行服务发现时只会获取正常运行的服务端配置信息。因此, 使用 ZooKeeper 也能同时满足高可用要求与以上第二点特性。

与 gRPC 这类 RPC 框架类似, 我们希望首先能定义出 RPC 接口, 并基于这个接口来开发服务端与客户端。也就是说, 在我们自行开发的 RPC 框架中同样也能做到面向接口编程, 这是第三点特性。如果我们使用 Protocol Buffers 来实现, 那么就需要编写 .proto 文件来定义 RPC 接口, 可通过 Maven 来生成 RPC 调用所需的 Stub 代码库。其实这样是为了做到对象的序列化与

反序列化，通过自动生成的代码，将这一既复杂又烦琐的过程给封装起来。幸运的是，我们可使用 Protostuff 将此过程变得完全透明，也就是说，我们不再编写 .proto 文件，也无须通过 Protocol Buffers 来自动生成 Stub 代码库，我们只需编写 Java 接口即可。整个 RPC 调用过程中，传输数据的序列化与反序列化工作交给 Protostuff 来完成，Protostuff 完全支持 Protocol Buffers。

关于 Protostuff 的更多信息，可通过它的官网进行学习，如图 4-8 所示。

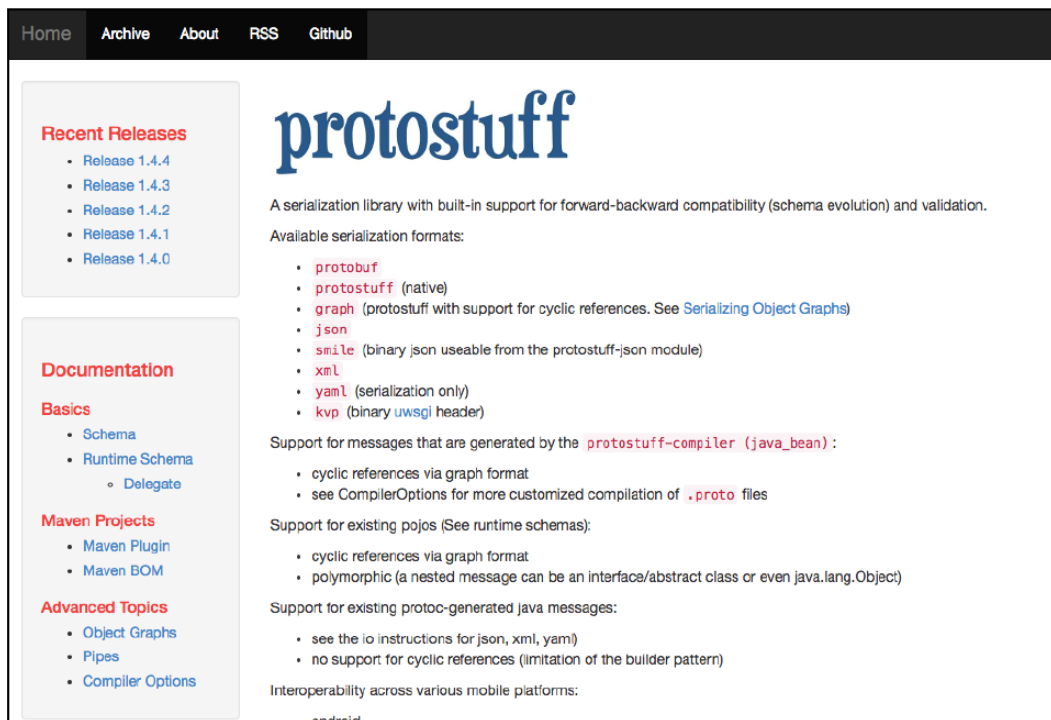


图 4-8 Protostuff 官网。

Protostuff 官网：<http://www.protostuff.io/>。

通过图 4-9 可以清晰地看出我们即将搭建的 RPC 框架的核心架构，以及应用程序与该框架之间的依赖关系。

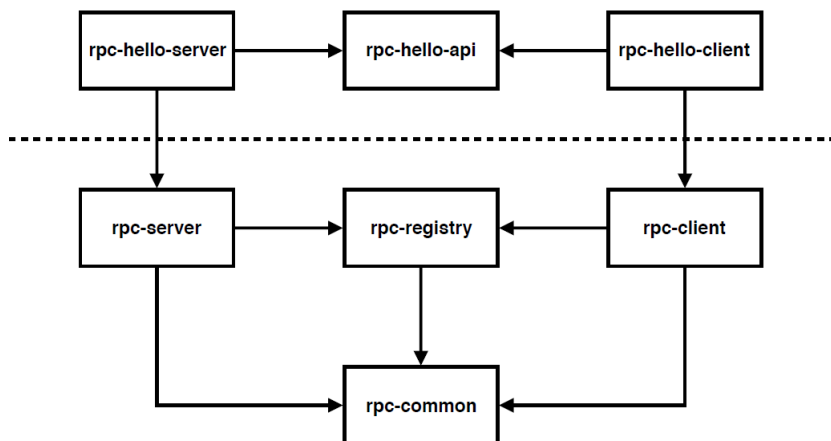


图 4-9 RPC 框架结构图

以上 RPC 框架结构图分为上下两部分（用虚线分割），上部分是一个示例应用程序，下部分是 RPC 框架及其内部模块结构。

对于上部分的示例应用程序而言，分为以下三个模块。

- （1）rpc-hello-api：包含 RPC 接口定义，即相关的 Java 接口。
- （2）rpc-hello-server：包含 RPC 服务端相关代码，即 RPC 接口的具体 Java 实现。
- （3）rpc-hello-client：包含 RPC 客户端相关代码，即调用 RPC 接口的使用方法。

对于下部分的 RPC 框架而言，分为以下四个模块。

- （1）rpc-server：包含启动 RPC 服务的框架代码，对 Netty 进行了封装。
- （2）rpc-client：包含调用 RPC 请求与处理 RPC 响应的框架代码，同样对 Netty 进行了封装。
- （3）rpc-registry：包含注册与发现 RPC 服务的框架代码，对 ZooKeeper 进行了封装。
- （4）rpc-common：包含以上模块的公共代码，比如对 Protostuff 序列化与序列化的过程进行了封装。

对于最核心的 rpc-server、rpc-client 和 rpc-registry 三个模块而言，接下来有必要对它们之间的交互过程进行相关说明。

当 RPC 服务端（rpc-server）启动后，会自动将其服务名称（service）与服务端所在主机名与端口号（host:port）注册到服务注册中心（rpc-registry），该过程称为“服务注册”，如图 4-10 所示。

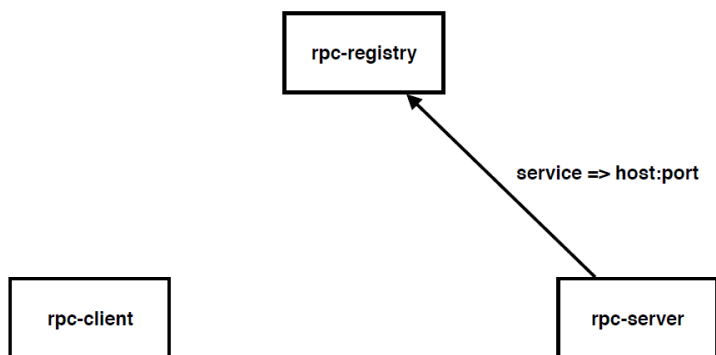


图 4-10 RPC 的“服务注册”过程

在 RPC 客户端发送 RPC 请求并调用 RPC 服务端之前，需要先通过服务名称从服务注册中心中获取相应的主机名与端口号，该过程称为“服务发现”，如图 4-11 所示。

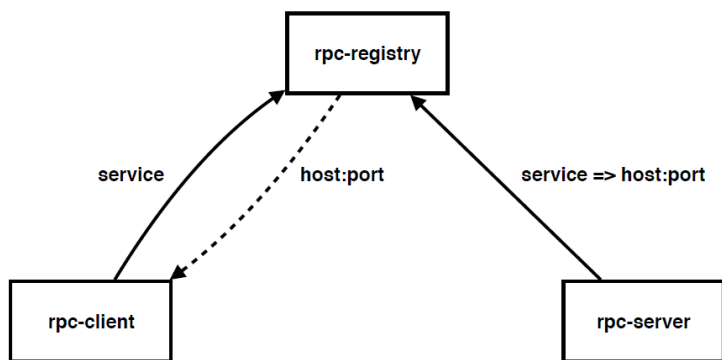


图 4-11 RPC 的“服务发现”过程

此时，RPC 客户端可通过主机名与端口号对 RPC 服务端发送 RPC 请求（request），该请求以同步方式执行，随后客户端将等待获取 RPC 服务端所返回的 RPC 响应（response），该过程称为“服务调用”，如图 4-12 所示。

可见，RPC 框架需要具备以上三种基本能力，即服务注册、服务发现、服务调用，这三种基本能力将通过以上三个核心模块来实现。

为了做到从零开始，我们接下来将对以上提到的几个模块，分别用 Maven 来搭建代码框架。

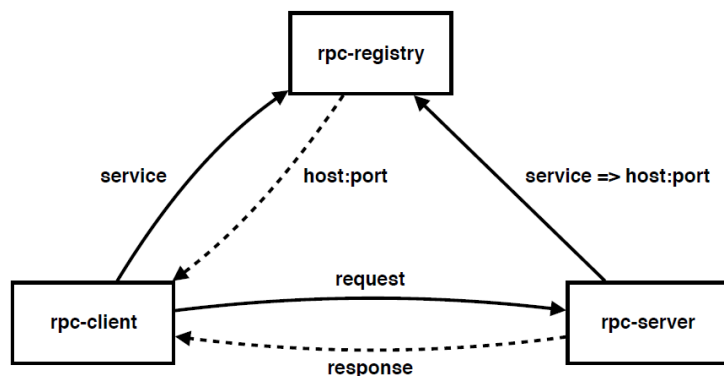


图 4-12 RPC 的“服务调用”过程

4.3.2 搭建模块代码框架

我们从底向上开始搭建整个代码框架，先搭建基础模块，随后搭建依赖于基础模块的高层模块。

第一步：搭建 rpc-common 模块。

以下是 rpc-common 模块的 pom.xml 配置文件的具体内容，提供了该框架需要用到的外部模块（或称为“三方库”）。

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>demo.msa</groupId>
    <artifactId>rpc-common</artifactId>
    <version>1.0.0</version>

    <dependencies>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>

```

```
        <version>1.7.25</version>
    </dependency>
    <dependency>
        <groupId>org.apache.commons</groupId>
        <artifactId>commons-lang3</artifactId>
        <version>3.5</version>
    </dependency>
    <dependency>
        <groupId>org.apache.commons</groupId>
        <artifactId>commons-collections4</artifactId>
        <version>4.1</version>
    </dependency>
    <dependency>
        <groupId>io.netty</groupId>
        <artifactId>netty-all</artifactId>
        <version>4.1.11.Final</version>
    </dependency>
    <dependency>
        <groupId>io.protostuff</groupId>
        <artifactId>protostuff-core</artifactId>
        <version>1.5.4</version>
    </dependency>
    <dependency>
        <groupId>io.protostuff</groupId>
        <artifactId>protostuff-runtime</artifactId>
        <version>1.5.4</version>
    </dependency>
    <dependency>
        <groupId>org.objenesis</groupId>
        <artifactId>objenesis</artifactId>
        <version>2.5.1</version>
    </dependency>
</dependencies>

</project>
```

下面，对所用到的相关模块进行简要说明。

- (1) **slf4j-api**: SLF4J 日志框架模块，用于在代码中输出应用程序的相关日志。

- (2) commons-lang3: Apache Commons 所提供的 Java 工具包, 包括 StringUtils 等工具类。
- (3) commons-collections4: Apache Commons 所提供的集合工具包, 包括 CollectionUtils 与 MapUtils 等工具类。
- (4) netty-all: Netty 模块, 提供 NIO 通信所需的 API, 它是整个 RPC 框架的基础。
- (5) protostuff-core: Protostuff 核心模块, 提供对象的序列化与反序列化功能。
- (6) protostuff-runtime: Protostuff 运行时模块, 用于生成所需的 Protostuff Schema 对象。
- (7) objenesis: 提供比 JDK 更加高效的反射功能, 可根据 Java 类来创建对应的实例对象, 在对象反序列化时需要用到它。

第二步: 搭建 rpc-registry 模块。

以下是 rpc-registry 模块的 pom.xml 配置文件的具体内容, 该模块依赖于 rpc-common 内部模块, 并依赖于 zookeeper、zkclient、spring-context 等外部模块。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>demo.msa</groupId>
    <artifactId>rpc-registry</artifactId>
    <version>1.0.0</version>

    <dependencies>
        <dependency>
            <groupId>demo.msa</groupId>
            <artifactId>rpc-common</artifactId>
            <version>1.0.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.zookeeper</groupId>
            <artifactId>zookeeper</artifactId>
            <version>3.4.10</version>
            <exclusions>
```

```

        <exclusion>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>com.101tec</groupId>
    <artifactId>zkclient</artifactId>
    <version>0.10</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>4.3.7.RELEASE</version>
    <scope>provided</scope>
</dependency>
</dependencies>

</project>

```

需要说明的是，此时需要排除 `zookeeper` 模块所依赖的 `slf4j-log4j12` 模块，因为 Spring Boot 默认使用 `Logback` 作为日志实现，而不是 `Log4j`。如果不排除该模块，那么启动 Spring Boot 应用程序时，将在控制台中输出相关的警告信息（SLF4J: Class path contains multiple SLF4J bindings.），此警告不会影响应用程序的正常运行。

此外，这里虽然依赖了 `spring-context` 模块，但此依赖的 `scope` 为 `provided`，表示该模块仅提供编译但无须参与打包，这是一个“弱依赖”关系。因为在应用程序中将依赖于特定 `spring-context` 模块，那时将决定该模块的具体版本号。默认情况下，`scope` 为 `compile`，表示“强依赖”关系，既提供编译又参与打包。

第三步：搭建 `rpc-server` 模块。

以下是 `rpc-server` 模块的 `pom.xml` 配置文件具体内容，该模块依赖于 `rpc-common`、`rpc-registry` 等内部模块，同样也依赖于 `spring-context` 外部模块。

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

```

```

        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

```

```

    <modelVersion>4.0.0</modelVersion>

    <groupId>demo.msa</groupId>
    <artifactId>rpc-server</artifactId>
    <version>1.0.0</version>

    <dependencies>
        <dependency>
            <groupId>demo.msa</groupId>
            <artifactId>rpc-common</artifactId>
            <version>1.0.0</version>
        </dependency>
        <dependency>
            <groupId>demo.msa</groupId>
            <artifactId>rpc-registry</artifactId>
            <version>1.0.0</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>4.3.7.RELEASE</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>

</project>

```

第四步：搭建 rpc-client 模块。

以下是 rpc-client 模块的 pom.xml 配置文件具体内容，该模块依赖于 rpc-common、rpc-registry 等内部模块，同样也依赖于 spring-context 外部模块。

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

```

```
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>demo.msa</groupId>
    <artifactId>rpc-client</artifactId>
    <version>1.0.0</version>

    <dependencies>
        <dependency>
            <groupId>demo.msa</groupId>
            <artifactId>rpc-common</artifactId>
            <version>1.0.0</version>
        </dependency>
        <dependency>
            <groupId>demo.msa</groupId>
            <artifactId>rpc-registry</artifactId>
            <version>1.0.0</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>4.3.7.RELEASE</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>

</project>
```

第五步：搭建 rpc-hello-api 模块。

以下是 rpc-hello-api 模块的 pom.xml 配置文件具体内容，该模块不依赖于任何其他模块，仅提供 RPC 接口，即使用 Java 编写的接口类。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
```



```

<modelVersion>4.0.0</modelVersion>

<groupId>demo.msa</groupId>
<artifactId>rpc-hello-api</artifactId>
<version>1.0.0</version>

</project>

```

第六步：搭建 rpc-hello-server 模块。

以下是 rpc-hello-server 模块的 pom.xml 配置文件具体内容，该模块基于 Spring Boot 框架，并依赖于 rpc-server、rpc-hello-api 等内部模块。

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>demo.msa</groupId>
    <artifactId>rpc-hello-server</artifactId>
    <version>1.0.0</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.2.RELEASE</version>
    </parent>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>demo.msa</groupId>

```

```

        <artifactId>rpc-server</artifactId>
        <version>1.0.0</version>
    </dependency>
    <dependency>
        <groupId>demo.msa</groupId>
        <artifactId>rpc-hello-api</artifactId>
        <version>1.0.0</version>
    </dependency>
</dependencies>

</project>

```

第七步：搭建 rpc-hello-client 模块。

以下是 rpc-hello-client 模块的 pom.xml 配置文件具体内容，该模块基于 Spring Boot 框架，并依赖于 rpc-client、rpc-hello-api 等内部模块。

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>demo.msa</groupId>
    <artifactId>rpc-hello-client</artifactId>
    <version>1.0.0</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.2.RELEASE</version>
    </parent>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>

```

```

    </dependency>
    <dependency>
        <groupId>demo.msa</groupId>
        <artifactId>rpc-client</artifactId>
        <version>1.0.0</version>
    </dependency>
    <dependency>
        <groupId>demo.msa</groupId>
        <artifactId>rpc-hello-api</artifactId>
        <version>1.0.0</version>
    </dependency>
</dependencies>

</project>

```

至此，所有模块的代码框架已搭建完毕，下面我们将着手开发 RPC 服务端与客户端，先从开发 RPC 服务端开始。

4.3.3 开发 RPC 服务端

我们首先定义 RPC 接口，随后实现该 RPC 接口的业务逻辑，最后完成 RPC 服务端的底层实现（包括服务扫描、服务启动、服务注册等特性）。通过以下五个步骤来完成 RPC 服务端开发，先从定义 RPC 接口开始。

第一步：定义 RPC 接口。

在 `rpc-hello-api` 模块中编写一个名为 `HelloService` 的 Java 接口，它表示 RPC 接口，该接口会在 `rpc-hello-server` 模块中进行实现，并在 `rpc-hello-client` 模块中进行调用。

```

package demo.msa.rpc.hello.api;

/**
 * RPC 接口
 *
 * @author huangyong
 * @since 1.0.0
 */
public interface HelloService {

```

```
String say(String name);  
}
```

该 RPC 接口包括一个简单的 `say()` 方法，输入一个 `String` 类型的 `name` 参数，输出一个 `String` 类型的返回值。

下面我们就来实现这个 RPC 接口。

第二步：实现 RPC 接口。

在 `rpc-hello-server` 模块中编写一个名为 `HelloServiceImpl` 的 Java 类，该类需实现在第一步中定义的 Java 接口。

```
package demo.msa.rpc.hello.server;  
  
import ...  
  
/**  
 * RPC 接口实现  
 *  
 * @author huangyong  
 * @since 1.0.0  
 */  
@RpcService(HelloService.class)  
public class HelloServiceImpl implements HelloService {  
  
    @Override  
    public String say(String name) {  
        return "hello " + name;  
    }  
}
```

该 RPC 接口实现中不仅完成了 RPC 接口的业务逻辑实现（简单地返回了一个“hello xxx”字符串），还通过 `@RpcService` 注解标示出该类是一个 RPC 服务类。该注解由 RPC 框架提供，用于在 RPC 服务端启动时，RPC 框架可自动扫描出带有 `@RpcService` 注解的 RPC 服务类，我们称该过程为“服务扫描”。

下面我们就来开发 RPC 框架的服务端部分，首先从 `@RpcService` 注解开始。

第三步：开发 RPC 服务端。

在 `rpc-server` 模块中编写一个名为 `RpcService` 的 Java 注解，该注解目前仅有一个名为 `value` 的属性，用于指定对应的 RPC 服务接口对应的 Class 对象。

```
package demo.msa.rpc.server;

import ...

/**
 * RPC 服务注解（标注在服务实现类上）
 *
 * @author huangyong
 * @since 1.0.0
 */
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Service
public @interface RpcService {

    /**
     * 服务接口类
     */
    Class<?> value();
}
```

该 `@RpcService` 注解上带有 Spring 框架提供的 `@Service` 注解，说明该注解具备 `@Service` 注解的特性。也就是说，凡是被 `@RpcService` 注解标示的类，都将被 Spring 框架扫描到，并通过反射创建出该类的实例，随后将其添加到 Spring IoC 容器内部的 `ApplicationContext` 对象中进行管理。因此，我们可以在 Spring 框架启动时，通过 `ApplicationContext` 对象来获取带有 `@RpcService` 注解的服务实例，从而建立服务名称与服务实例之间的映射关系，便于后续从 RPC 请求中获取服务名称，并直接获取服务实例，最终通过反射来调用服务实例的目标方法。

可见，服务扫描是整个 RPC 服务端框架需要完成的首要任务，我们的最终目标是开发一个 RPC 服务器，用于发布服务端所实现的 RPC 服务。

在 `rpc-server` 模块中创建一个 `RpcServer` 类，该类实现于 Spring 框架所提供的 `org.springframework.context.ApplicationContextAware` 接口，并在 `RpcServer` 类中重写该接口的 `setApplicationContext()` 方法，此时可获取 `ApplicationContext` 对象，进而完成我们想要实现的“服务扫描”特性。

```
package demo.msa.rpc.server;

import ...

/**
 * RPC 服务器（用于发布 RPC 服务）
 *
 * @author huangyong
 * @since 1.0.0
 */
@Component
public class RpcServer implements ApplicationContextAware {

    /**
     * 存放服务名称与服务实例之间的映射关系
     */
    private Map<String, Object> handlerMap = new HashMap<>();

    @Override
    public void setApplicationContext(ApplicationContext ctx) throws
BeansException {
        // 扫描带有 @RpcService 注解的服务类
        Map<String, Object> serviceBeanMap = ctx.getBeansWithAnnotation
(RpcService.class);
        if (CollectionUtil.isEmpty(serviceBeanMap)) {
            for (Object serviceBean : serviceBeanMap.values()) {
                RpcService rpcService = serviceBean.getClass().getAnnotation
(RpcService.class);
                String serviceName = rpcService.value().getName();
                handlerMap.put(serviceName, serviceBean);
            }
        }
    }
}
```

我们通过调用 `ApplicationContext` 对象的 `getBeansWithAnnotation()` 方法来获取带有 `@RpcService` 注解的服务实例，并通过一个循环来遍历每个服务实例，获取服务名称，并将服

务名称与服务实例建立映射关系，最终将该映射关系存放在一个 `Map<String, Object>` 类型的 `handlerMap` 对象中，以供后续使用。

在 RPC 服务器除了需要“服务扫描”，还需要完成“服务启动”与“服务注册”这两个重要的任务。前者已经实现，后两者需要通过实现 Spring 框架的 `org.springframework.beans.factory.InitializingBean` 接口，并在该接口的 `afterPropertiesSet()` 方法中完成 RPC 服务器的初始化工作。

```
package demo.msa.rpc.server;

import ...

/**
 * RPC 服务器（用于发布 RPC 服务）
 *
 * @author huangyong
 * @since 1.0.0
 */
@Component
public class RpcServer implements ApplicationContextAware, InitializingBean {

    private static final Logger logger = LoggerFactory.getLogger(RpcServer.class);

    @Value("${rpc.port}")
    private int port;

    @Autowired
    private ServiceRegistry serviceRegistry;

    /**
     * 存放服务名称与服务实例之间的映射关系
     */
    private Map<String, Object> handlerMap = new HashMap<>();

    @Override
    public void setApplicationContext(ApplicationContext ctx) throws BeansException {
        ...
    }
}
```

```

    }

    @Override
    public void afterPropertiesSet() throws Exception {
        EventLoopGroup group = new NioEventLoopGroup();
        EventLoopGroup childGroup = new NioEventLoopGroup();
        try {
            // 启动 RPC 服务
            ServerBootstrap bootstrap = new ServerBootstrap();
            bootstrap.group(group, childGroup);
            bootstrap.channel(NioServerSocketChannel.class);
            bootstrap.childHandler(new ChannelInitializer<SocketChannel>() {
                @Override
                public void initChannel(SocketChannel channel) throws Exception {
                    ChannelPipeline pipeline = channel.pipeline();
                    pipeline.addLast(new RpcDecoder(RpcRequest.class)); // 解码 RPC 请求
                    pipeline.addLast(new RpcEncoder(RpcResponse.class)); // 编码 RPC 响应
                    pipeline.addLast(new RpcServerHandler(handlerMap)); // 处理 RPC 请求
                }
            });
            ChannelFuture future = bootstrap.bind(port).sync();
            logger.debug("server started, listening on {}", port);
            // 注册 RPC 服务地址
            String serviceAddress = InetAddress.getLocalHost().getHostAddress()
+ ":" + port;
            for (String interfaceName : handlerMap.keySet()) {
                serviceRegistry.register(interfaceName, serviceAddress);
                logger.debug("register service: {} => {}", interfaceName,
serviceAddress);
            }
            // 释放资源
            future.channel().closeFuture().sync();
        } catch (Exception e) {
            logger.error("server exception", e);
        } finally {
            // 关闭 RPC 服务
            childGroup.shutdownGracefully();
        }
    }
}

```



```

        group.shutdownGracefully();
    }
}
}

```

在 `InitializingBean` 接口的 `afterPropertiesSet()` 方法中，我们使用 `Netty` 提供的 API 开发了一个基于 NIO 通信方式的 RPC 服务器。在创建 `ServerBootstrap` 对象时，添加了两个 `EventLoopGroup` 对象（`bossGroup` 与 `childGroup`），`bossGroup` 用于控制并管理相关 `childGroup`，`childGroup` 用于处理相关 RPC 请求。在 `SocketChannel` 中需要建立一个流水线（`Pipeline`），先后完成解码 RPC 请求、编码 RPC 响应、处理 RPC 请求等任务。通过调用 `ServerBootstrap` 对象的 `bind()` 方法来启动 RPC 服务器，此时需调用 `sync()` 方法，确保该操作以同步的方式来执行。也就是说，必须等待 RPC 服务器启动完毕后，才会执行后续的代码，此时返回了一个 `ChannelFuture` 对象。随后需要获取本地主机名（`host`），并通过配置传入的端口号（`port`），将其拼接成一个 RPC 服务地址字符串（`host:port`），并调用 `ServiceRegistry` 对象的 `register()` 方法来完成“服务注册”操作。随后我们需要以同步的方式来关闭 `ChannelFuture` 对象，此时 RPC 服务器将等待自身应用程序退出，并释放所占用的资源。最终分别调用 `bossGroup` 与 `childGroup` 的 `shutdownGracefully()` 方法来关闭 RPC 服务器。

那么，RPC 请求（`RpcRequest`）中需要封装哪些信息呢？

在 `rpc-common` 模块中，创建一个 `RpcRequest` 类，该类提供了一些 RPC 请求所需的相关属性。

```

package demo.msa.rpc.common.bean;

/**
 * 封装 RPC 请求
 *
 * @author huangyong
 * @since 1.0.0
 */
public class RpcRequest {

    /**
     * 请求编号
     */
    private String requestId;
}

```

```
/**
 * 接口名称
 */
private String interfaceName;

/**
 * 方法名称
 */
private String methodName;

/**
 * 参数类型
 */
private Class<?>[] parameterTypes;

/**
 * 参数对象
 */
private Object[] parameters;

// 省略 getter/setter 方法
}
```

可见，每个 `RpcRequest` 对象包括一个请求编号（`requestId`），还包括一组目标接口的特征属性。`requestId` 可在程序中自动生成，只需确保它能唯一识别每个 RPC 请求即可。

同样，RPC 响应（`RpcResponse`）中也需要有一个 `requestId`，可以与 `RpcRequest` 形成一对“请求—响应”关系。因为在我们的 RPC 框架中需要识别哪个 `RpcResponse` 来自于哪个 `RpcRequest`，所以在客户端中需要通过 `requestId` 来识别返回的 `RpcResponse`。

在 `rpc-common` 模块中创建一个 `RpcResponse` 类，该类提供了一些 RPC 响应所需的相关属性。

```
package demo.msa.rpc.common.bean;

/**
 * 封装 RPC 响应
 *
 * @author huangyong
```

```
* @since 1.0.0
*/
public class RpcResponse {

    /**
     * 请求编号
     */
    private String requestId;

    /**
     * 异常信息
     */
    private Exception exception;

    /**
     * 响应结果
     */
    private Object result;

    /**
     * 是否带有异常
     */
    public boolean hasException() {
        return exception != null;
    }

    // 省略 getter/setter 方法
}
```

`RpcResponse` 除了包括 `requestId`，还要考虑两种情况，一种是调用成功，客户端可从 `result` 属性中获取响应结果；另一种是调用失败，客户端可从 `exception` 属性中获取异常信息。此外，还提供了名为 `hasException()` 的工具方法，用于判断响应中是否带有异常信息，从而判断调用是否失败。

`RpcRequest` 与 `RpcResponse` 无须实现 JDK 提供的 `java.io.Serializable` 接口，因为我们所开发的 RPC 框架不会使用 JDK 默认提供的序列化与反序列化机制，而是使用 `Protostuff` 来实现同样类似的功能。此外，由于 `Netty` 框架只允许在 NIO 通道（`Channel`）中传输自己提供的 `ByteBuf` 对象，其他类型对象必须进行“编码”，才能在 `Channel` 中进行传输。此外，从 `Channel` 中获取

的 `ByteBuf` 对象还需进行“解码”，才能转换为我们需要的原始类型对象。Netty 已经为我们提供了一套非常好用的编码与解码框架，我们只需通过简单的扩展就能将 `Protostuff` 集成到 Netty 框架中，从而实现基于 `Protostuff` 的编码与解码，即序列化与反序列化操作。

在 `rpc-common` 模块中，创建一个 `RpcEncoder` 类，它是 RPC 通信过程的编码器，我们只需传入一个待编码的数据类型，在编码过程中就会将此数据类型进行序列化操作，将 `Object` 对象序列化为 `byte[]` 数据。在 Netty 中，将 `byte[]` 封装到 `ByteBuf` 对象中，并将 `ByteBuf` 对象放在 `Channel` 中进行传输。

```
package demo.msa.rpc.common.codec;

import ...

/**
 * RPC 编码器
 *
 * @author huangyong
 * @since 1.0.0
 */
public class RpcEncoder extends MessageToByteEncoder {

    private Class<?> genericClass;

    public RpcEncoder(Class<?> genericClass) {
        this.genericClass = genericClass;
    }

    @Override
    public void encode(ChannelHandlerContext ctx, Object in, ByteBuf out)
        throws Exception {
        if (genericClass.isInstance(in)) {
            byte[] data = SerializationUtil.serialize(in); // 序列化
            out.writeInt(data.length);
            out.writeBytes(data);
        }
    }
}
```

在 `rpc-common` 模块中创建一个 `RpcDecoder` 类，此时需传入待解码的数据类型，最终将 `ByteBuf` 对象反序列化为 `Object` 对象。

```
package demo.msa.rpc.common.codec;

import ...

/**
 * RPC 解码器
 *
 * @author huangyong
 * @since 1.0.0
 */
public class RpcDecoder extends ByteToMessageDecoder {

    private Class<?> genericClass;

    public RpcDecoder(Class<?> genericClass) {
        this.genericClass = genericClass;
    }

    @Override
    public void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out)
        throws Exception {
        if (in.readableBytes() < 4) {
            return;
        }
        in.markReaderIndex();
        int dataLength = in.readInt();
        if (in.readableBytes() < dataLength) {
            in.resetReaderIndex();
            return;
        }
        byte[] data = new byte[dataLength];
        in.readBytes(data);
        out.add(SerializationUtil.deserialize(data, genericClass)); // 反序列化
    }
}
```

在 RpcEncoder 与 RpcDecoder 中，用到了一个名为 SerializationUtil 的工具类，该类基于 Protostuff 实现，提供了底层的序列化与反序列化操作。

在 rpc-common 模块中，创建一个 SerializationUtil 工具类，封装 Protostuff 所提供的序列化与反序列化相关代码。

```
package demo.msa.rpc.common.util;

import ...

/**
 * 序列化工具类（基于 Protostuff 实现）
 *
 * @author huangyong
 * @since 1.0.0
 */
public class SerializationUtil {

    private static final Map<Class<?>, Schema<?>> cachedSchema = new
ConcurrentHashMap<>();

    private static final Objenesis objenesis = new ObjenesisStd(true);

    /**
     * 序列化（对象 -> 字节数组）
     */
    @SuppressWarnings("unchecked")
    public static <T> byte[] serialize(T obj) {
        Class<T> cls = (Class<T>) obj.getClass();
        LinkedBuffer buffer = LinkedBuffer.allocate(LinkedBuffer.DEFAULT_
BUFFER_SIZE);
        try {
            Schema<T> schema = getSchema(cls);
            return ProtostuffIOUtil.toByteArray(obj, schema, buffer);
        } catch (Exception e) {
            throw new IllegalStateException(e.getMessage(), e);
        } finally {
            buffer.clear();
        }
    }
}
```

```

    }
}

/**
 * 反序列化 ( 字节数组 -> 对象 )
 */
public static <T> T deserialize(byte[] data, Class<T> cls) {
    try {
        T message = objenesis.newInstance(cls);
        Schema<T> schema = getSchema(cls);
        ProtostuffIOUtil.mergeFrom(data, message, schema);
        return message;
    } catch (Exception e) {
        throw new IllegalStateException(e.getMessage(), e);
    }
}

@SuppressWarnings("unchecked")
private static <T> Schema<T> getSchema(Class<T> cls) {
    Schema<T> schema = (Schema<T>) cachedSchema.get(cls);
    if (schema == null) {
        schema = RuntimeSchema.createFrom(cls);
        cachedSchema.put(cls, schema);
    }
    return schema;
}
}

```

在 `RpcServer` 类中，除了对 `ChannelPipeline` 添加了 `RpcDecoder` 与 `RpcEncoder` 编解码处理器，我们还添加了一个核心的处理器，用它来处理每个 RPC 请求，即 RPC 服务端的请求处理器。

在 `rpc-server` 模块中，创建一个 `RpcServerHandler` 类，该类继承于 Netty 提供的 `SimpleChannelInboundHandler` 抽象类，此时需将 `RpcRequest` 类作为泛型传入其中。

```

package demo.msa.rpc.server;

import ...

```

```

/**
 * RPC 服务端处理器（用于处理 RPC 请求）
 *
 * @author huangyong
 * @since 1.0.0
 */
public class RpcServerHandler extends SimpleChannelInboundHandler<RpcRequest> {

    @Override
    public void channelRead0(ChannelHandlerContext ctx, RpcRequest request)
throws Exception {
        }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
        }
}

```

我们重写了 `SimpleChannelInboundHandler` 抽象类的两个方法：`channelRead0()`方法用于从 `Channel` 中读取数据，该方法的参数列表中带有一个 `RpcRequest` 参数，它就是所继承的抽象类中指定的 `RpcRequest` 泛型参数；`exceptionCaught()`方法用于捕获 RPC 通信过程中所出现的异常。

接下来，我们将分别实现以上两个方法，先从比较简单的 `exceptionCaught()`方法开始。

```

package demo.msa.rpc.server;

import ...

/**
 * RPC 服务端处理器（用于处理 RPC 请求）
 *
 * @author huangyong
 * @since 1.0.0
 */
public class RpcServerHandler extends SimpleChannelInboundHandler<RpcRequest> {

    private static final Logger logger = LoggerFactory.getLogger
(RpcServerHandler.class);

```



```

        @Override
        public void channelRead0(ChannelHandlerContext ctx, RpcRequest request)
throws Exception {
        }

        @Override
        public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
            logger.error("server caught exception", cause);
            ctx.close();
        }
    }
}

```

这个方法实现非常简单，只需向日志对象里输出一句错误信息，然后关闭 `ChannelHandlerContext` 对象即可，实际上此时关闭的是服务端与客户端的 `Channel` 连接。

下面我们继续完成 `channelRead0()` 方法，它的实现逻辑比较复杂，需要从 `RpcRequest` 参数中获取相关属性，并执行 Java 反射调用，最终将生成一个 `RpcResponse` 对象，并将其写入 `Channel` 中，最终才能被客户端获取该 `RpcResponse` 对象。

```

package demo.msa.rpc.server;

import ...

/**
 * RPC 服务端处理器（用于处理 RPC 请求）
 *
 * @author huangyong
 * @since 1.0.0
 */
public class RpcServerHandler extends SimpleChannelInboundHandler<RpcRequest> {

    private static final Logger logger = LoggerFactory.getLogger
(RpcServerHandler.class);

    /**
     * 存放服务名称与服务实例之间的映射关系
     */
    private final Map<String, Object> handlerMap;

```

```
public RpcServerHandler(Map<String, Object> handlerMap) {
    this.handlerMap = handlerMap;
}

@Override
public void channelRead0(ChannelHandlerContext ctx, RpcRequest request)
throws Exception {
    // 创建 RPC 响应对象
    RpcResponse response = new RpcResponse();
    response.setRequestId(request.getRequestId());
    try {
        // 处理 RPC 请求成功
        Object result = handle(request);
        response.setResult(result);
    } catch (Exception e) {
        // 处理 RPC 请求失败
        response.setException(e);
        logger.error("handle result failure", e);
    }
    // 写入 RPC 响应对象（写入完毕后立即关闭与客户端的连接）
    ctx.writeAndFlush(response).addListener(ChannelFutureListener.CLOSE);
}

private Object handle(RpcRequest request) throws Exception {
    // 获取服务实例
    String serviceName = request.getInterfaceName();
    Object serviceBean = handlerMap.get(serviceName);
    if (serviceBean == null) {
        throw new RuntimeException(String.format("can not find service bean
by key: %s", serviceName));
    }
    // 获取反射调用所需的变量
    Class<?> serviceClass = serviceBean.getClass();
    String methodName = request.getMethodName();
    Class<?>[] parameterTypes = request.getParameterTypes();
    Object[] parameters = request.getParameters();
    // 执行反射调用
    Method method = serviceClass.getMethod(methodName, parameterTypes);
```

```

        method.setAccessible(true);
        return method.invoke(serviceBean, parameters);
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
        ...
    }
}

```

我们首先定义了一个名为 `handlerMap` 的 `Map<String, Object>` 对象，用于存放服务名称与服务实例之间的映射关系，该映射关系实际上是从 `RpcServer` 类中传递进来的，这样做只是为了对该对象中的数据进行引用，确保两个类能共享同一份数据。在 `channelRead0()` 方法中，首先创建了一个 `RpcResponse` 对象，并将 `requestId` 初始化到该对象中，确保 `RpcResponse` 与 `RpcRequest` 可形成一对“请求—响应”关系。随后调用了自行封装的 `handle()` 方法，处理最核心的调用过程。在 `handle()` 方法中，我们需要先从 `RpcRequest` 中获取 `serviceName`，再从 `handlerMap` 中根据 `serviceName` 来获取 `serviceBean`，随后通过 Java 反射方式来调用 `serviceBean` 中的目标方法，并将反射调用结果分两种情况进行判断：若调用成功，则将调用结果放入 `RpcResponse` 对象中；若调用失败，则将相关异常放入 `RpcResponse` 对象中。最后将 `RpcResponse` 对象通过 `ChannelHandlerContext` 对象写入 `Channel` 中，并在写入完毕后立即关闭与客户端的连接，此时用到了 Netty 提供的监听回调技术。

当服务启动成功后，我们需将服务配置信息注册到基于 ZooKeeper 的“服务注册中心”中。在 `RpcServer` 中，我们注入了 `ServiceRegistry` 对象，用它来实现服务注册功能。

第四步：实现服务注册功能。

在 `rpc-registry` 模块中，创建一个 `ServiceRegistry` 类，用于提供服务注册功能。

```

package demo.msa.rpc.registry;

import ...

/**
 * 服务注册
 *
 * @author huangyong
 * @since 1.0.0
 */

```

```
@Component
public class ServiceRegistry {

    private static final Logger logger = LoggerFactory.getLogger
(ServiceRegistry.class);

    @Value("${rpc.registry-address}")
    private String zkAddress;

    private ZkClient zkClient;

    @PostConstruct
    public void init() {
        // 创建 ZooKeeper 客户端
        zkClient = new ZkClient(zkAddress, Constant.ZK_SESSION_TIMEOUT,
Constant.ZK_CONNECTION_TIMEOUT);
        logger.debug("connect to zookeeper");
    }

    public void register(String serviceName, String serviceAddress) {
        // 创建 registry 节点（持久）
        String registryPath = Constant.ZK_REGISTRY_PATH;
        if (!zkClient.exists(registryPath)) {
            zkClient.createPersistent(registryPath);
            logger.debug("create registry node: {}", registryPath);
        }
        // 创建 service 节点（持久）
        String servicePath = registryPath + "/" + serviceName;
        if (!zkClient.exists(servicePath)) {
            zkClient.createPersistent(servicePath);
            logger.debug("create service node: {}", servicePath);
        }
        // 创建 address 节点（临时）
        String addressPath = servicePath + "/address-";
        String addressNode = zkClient.createEphemeralSequential(addressPath,
serviceAddress);
        logger.debug("create address node: {}", addressNode);
    }
}
```

在 `ServiceRegistry` 类中，有一个 `init()` 方法，该方法带有 `@PostConstruct` 注解，说明该方法将在对象创建完毕后（即构造方法执行后）被 Spring 框架调用。在 `init()` 方法中，我们使用 `ZkClient` 连接了 ZooKeeper 服务器，并通过 `register()` 方法来实现服务注册功能。我们首先创建了一个名为 `registry` 的根节点，该节点是持久类型的，因为它下面还有子节点。随后创建了一个包含服务名称的 `service` 节点，该节点仍然是持久的。最后创建了一个对应服务地址的 `address` 节点，该节点是临时类型的，因为它不再有子节点，而且当客户端与服务端断开连接后，ZooKeeper 的“心跳检测”失败，此时将自动移出该节点。

在 `rpc-hello-server` 模块中，创建一个 `application.properties` 配置文件（该文件位于 `src/main/resources` 目录下），我们在该文件中添加以下两个配置项。

```
rpc.port=8000
rpc.registry-address=localhost:2181
```

`rpc.port` 配置项表示 RPC 服务端运行所需监听的端口号，`rpc.registry-address` 配置项表示 ZooKeeper 注册中心的 IP 地址与端口号。

在 `ServiceRegistry` 类中涉及了一些常量，我们统一将它们封装在 `rpc-registry` 模块的 `Constant` 常量接口中。

```
package demo.msa.rpc.registry;

/**
 * 相关常量
 *
 * @author huangyong
 * @since 1.0.0
 */
public interface Constant {

    int ZK_SESSION_TIMEOUT = 5000;
    int ZK_CONNECTION_TIMEOUT = 1000;

    String ZK_REGISTRY_PATH = "/registry";
}
```

目前服务注册功能已开发完毕，我们接下来就可以启动 RPC 服务器了。

第五步：启动 RPC 服务器。

在启动 RPC 服务器之前，首先要做的是启动 ZooKeeper 服务器，因为 RPC 服务器在启动时需要连接 ZooKeeper 服务器。

```
docker run \  
--rm \  
-p 2181:2181 \  
--name zookeeper \  
zookeeper
```

随后，在 `rpc-hello-server` 模块中，创建一个 Spring Boot 应用程序启动类 `HelloServerApplication`，我们直接运行它就能启动 RPC 服务器。

```
package demo.msa.rpc.hello.server;  
  
import ...  
  
@SpringBootApplication(scanBasePackages = "demo.msa.rpc")  
public class HelloServerApplication {  
  
    public static void main(String[] args) {  
        SpringApplication.run(HelloServerApplication.class, args);  
    }  
}
```

需要注意的是，我们必须指定 `scanBasePackages` 作为 Spring Boot 类扫描的基础包名，否则应用程序无法成功加载 RPC 服务端框架。

当启动 RPC 服务器成功后，我们可在控制台中看到如下信息。

```
2017-05-23 19:23:20.044 DEBUG 3114 --- [          main]  
demo.msa.rpc.registry.ServiceRegistry : connect to zookeeper  
2017-05-23 19:23:20.215 DEBUG 3114 --- [          main]  
demo.msa.rpc.server.RpcServer          : server started, listening on 8000  
2017-05-23 19:23:20.248 DEBUG 3114 --- [          main]  
demo.msa.rpc.registry.ServiceRegistry : create address node:  
/registry/demo.msa.rpc.hello.api.HelloService/address-0000000006  
2017-05-23 19:23:20.248 DEBUG 3114 --- [          main]  
demo.msa.rpc.server.RpcServer          : register service:  
demo.msa.rpc.hello.api.HelloService => 10.18.29.40:8000
```

此时说明 RPC 服务器启动完毕，在控制台中输出了相应的日志信息，接下来我们可通过 ZooKeeper 客户端来查看已注册的服务信息。

```
docker run \  
--rm \  
-it \  
--link zookeeper \  
zookeeper \  
zkCli.sh -server zookeeper
```

连接 ZooKeeper 成功后，可在 ZooKeeper 客户端命令行中输入以下 get 命令，获取对应的节点信息。

```
[zk: zookeeper(CONNECTED) 0] get /registry/demo.msa.rpc.hello.api.  
HelloService/address-0000000000  
10.18.29.40:8000  
cZxid = 0x5b  
ctime = Tue May 23 11:23:20 GMT 2017  
mZxid = 0x5b  
mtime = Tue May 23 11:23:20 GMT 2017  
pZxid = 0x5b  
cversion = 0  
dataVersion = 0  
aclVersion = 0  
ephemeralOwner = 0x15c274409440029  
dataLength = 23  
numChildren = 0
```

以上输出信息的第一行就是服务端所在的 IP 与端口号，它就是客户端连接服务端所需的服务配置，随后我们通过 ZooKeeper 客户端实现“服务发现”功能，来获取这些服务配置，客户端通过该服务配置与服务端建立 RPC 连接，最终调用远程服务接口中的目标方法，现在我们就来开发 RPC 客户端。

4.3.4 开发 RPC 客户端

由于客户端只能调用本地的 RPC 接口，无法访问远程的 RPC 实现，因此我们可以想到的解决方案是，当客户端调用本地 RPC 接口时，通过 RPC 接口来创建一个动态代理对象，并通

过这个动态代理对象与服务端建立 RPC 连接，从而执行远程调用。通过以下四个步骤来完成 RPC 客户端开发，先从封装 RPC 客户端代理开始。

第一步：封装 RPC 客户端代理。

在 `rpc-client` 模块中，创建一个 `RpcClient` 类，我们将在该类中使用 JDK 动态代理技术实现 RPC 客户端。当调用代理接口的方法时，将发送 RPC 请求，此时会使用“服务发现”机制来获取服务端配置信息，也会使用 `Netty` 与客户端建立 NIO 通道，并返回最终的 RPC 响应结果。

```
package demo.msa.rpc.client;

import ...

/**
 * RPC 客户端（用于创建 RPC 服务代理）
 *
 * @author huangyong
 * @since 1.0.0
 */
@Component
public class RpcClient {

    private static final Logger logger = LoggerFactory.getLogger(RpcClient.class);

    @Autowired
    private ServiceDiscovery serviceDiscovery;

    /**
     * 存放请求编号与响应对象之间的映射关系
     */
    private ConcurrentMap<String, RpcResponse> responseMap = new
    ConcurrentHashMap<>();

    @SuppressWarnings("unchecked")
    public <T> T create(final Class<?> interfaceClass) {
        // 创建动态代理对象
        return (T) Proxy.newProxyInstance(
            interfaceClass.getClassLoader(),
            new Class<?>[] {interfaceClass},
```



```
new InvocationHandler() {
    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
    // 创建 RPC 请求对象
    RpcRequest request = new RpcRequest();
    request.setRequestId(UUID.randomUUID().toString());
    request.setInterfaceName(method.getDeclaringClass().getName());
    request.setMethodName(method.getName());
    request.setParameterTypes(method.getParameterTypes());
    request.setParameters(args);
    // 获取 RPC 服务地址
    String serviceName = interfaceClass.getName();
    String serviceAddress = serviceDiscovery.discover(serviceName);
    logger.debug("discover service: {} => {}", serviceName,
serviceAddress);
    if (StringUtil.isEmpty(serviceAddress)) {
        throw new RuntimeException("server address is empty");
    }
    // 从 RPC 服务地址中解析主机名与端口号
    String[] array = StringUtil.split(serviceAddress, ":");
    String host = array[0];
    int port = Integer.parseInt(array[1]);
    // 发送 RPC 请求
    RpcResponse response = send(request, host, port);
    if (response == null) {
        logger.error("send request failure", new IllegalStateException
("response is null"));
        return null;
    }
    if (response.hasException()) {
        logger.error("response has exception", response.getException());
        return null;
    }
    // 获取响应结果
    return response.getResult();
}
}
```

```

    );
}

private RpcResponse send(RpcRequest request, String host, int port) {
    EventLoopGroup group = new NioEventLoopGroup(1); // 单线程模式
    try {
        // 创建 RPC 连接
        Bootstrap bootstrap = new Bootstrap();
        bootstrap.group(group);
        bootstrap.channel(NioSocketChannel.class);
        bootstrap.handler(new ChannelInitializer<SocketChannel>() {
            @Override
            public void initChannel(SocketChannel channel) throws Exception {
                ChannelPipeline pipeline = channel.pipeline();
                pipeline.addLast(new RpcEncoder(RpcRequest.class)); // 编码 RPC 请求
                pipeline.addLast(new RpcDecoder(RpcResponse.class)); // 解码 RPC 响应
                pipeline.addLast(new RpcClientHandler(responseMap)); // 处理 RPC 响应
            }
        });
        ChannelFuture future = bootstrap.connect(host, port).sync();
        // 写入 RPC 请求对象
        Channel channel = future.channel();
        channel.writeAndFlush(request).sync();
        channel.closeFuture().sync();
        // 获取 RPC 响应对象
        return responseMap.get(request.getRequestId());
    } catch (Exception e) {
        logger.error("client exception", e);
        return null;
    } finally {
        // 关闭 RPC 连接
        group.shutdownGracefully();
        // 移除请求编号与响应对象之间的映射关系
        responseMap.remove(request.getRequestId());
    }
}
}
}

```

我们首先注入了 `ServiceDiscovery` 对象,它用于提供“服务发现”功能(后续将实现该功能)。还定义了一个 `ConcurrentMap<String, RpcResponse>` 类型的 `responseMap` 对象,用来存放请求编号(`requestId`)与响应对象(`RpcResponse`)之间的映射关系。由于客户端需要考虑线程安全问题,因此这里使用了 Java 并发包中的 `ConcurrentMap` 及其实现 `ConcurrentHashMap`。在 `create()` 方法中,我们使用了 JDK 提供的 `Proxy.newProxyInstance()` 方法来创建动态代理对象,此时需传入一个被代理的接口对象(`interfaceClass`),并通过强制类型转换为代理对象类型。在创建动态代理对象时,需要传入当前的类加载器(`ClassLoader`)、被代理的接口类型、`InvocationHandler` 对象。其中 `InvocationHandler` 可通过匿名内部对象的方式来创建,此时需实现 `InvocationHandler` 接口的 `invoke()` 方法。在该方法内部,我们创建了 RPC 请求对象,通过 `ServiceDiscovery` 对象获取了 RPC 服务地址,并调用自己封装的 `send()` 方法来发送 RPC 请求。调用 `send()` 方法结束后,将返回对应的 RPC 响应对象,此时需判断响应对象中的异常情况,从而获取最终的响应结果。在 `send()` 方法中,我们通过 Netty 提供的 API 编写了一个 RPC 客户端,与 RPC 服务端类似,需要设置 `EventLoopGroup` 对象,设置 NIO 通道,以及添加一系列通道处理器(`ChannelHandler`),从而创建 `Bootstrap` 对象。通过调用 `Bootstrap` 对象的 `bind()` 方法,以同步的方式来连接 RPC 服务端,此时需传入服务端所在的主机名与端口号。与服务端连接成功后,将返回 `ChannelFuture` 对象,通过调用该对象的 `writeAndFlush()` 方法,将 RPC 请求对象写入 NIO 通道中,随后以同步的方式关闭 `ChannelFuture` 对象,即关闭了与服务端的通道。最终优雅地关闭 RPC 连接,并移除请求编号与响应对象之间的映射关系。

需要注意的是,客户端每次调用本地 RPC 接口时,实际上都会创建一个动态代理对象,并与服务端进行一次 RPC 连接。在高并发的场景下,将造成客户端的资源浪费,因此这里有必要进行一些优化——可将每次创建的动态代理对象缓存起来,该优化请大家自行扩展。此外,请大家思考一下,此时为何在创建 `EventLoopGroup` 对象时,我们要通过单线程模式来创建呢?

在 `RpcClient` 类的 `send()` 方法中,我们向 `ChannelPipeline` 中添加了三个处理器,前两个分别用于编码 RPC 请求对象与解码 RPC 响应对象,最后一个 `RpcClientHandler` 用于处理从服务端获取的解码后的 RPC 响应对象。由于编码与解码过程我们已经在服务端开发中实现了,因此下面我们将精力集中在 `RpcClientHandler` 上。

在 `rpc-client` 模块中创建一个 `RpcClientHandler` 类,并编写以下代码。

```
package demo.msa.rpc.client;

import ...

/**
 * RPC 客户端处理器 (用于处理 RPC 响应)
 */
```

```
* @author huangyong
* @since 1.0.0
*/
public class RpcClientHandler extends SimpleChannelInboundHandler<RpcResponse> {

    private static final Logger logger = LoggerFactory.getLogger
(RpcClientHandler.class);

    /**
     * 存放请求编号与响应对象之间的映射关系
     */
    private ConcurrentMap<String, RpcResponse> responseMap;

    public RpcClientHandler(ConcurrentMap<String, RpcResponse> responseMap) {
        this.responseMap = responseMap;
    }

    @Override
    public void channelRead0(ChannelHandlerContext ctx, RpcResponse response)
throws Exception {
        // 建立请求编号与响应对象之间的映射关系
        responseMap.put(response.getRequestId(), response);
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
throws Exception {
        logger.error("client caught exception", cause);
        ctx.close();
    }
}
```

该类同样继承于 Netty 提供的 `SimpleChannelInboundHandler` 抽象类，同时传入了 `RpcResponse` 作为泛型参数，这意味着 `channelRead0()` 方法中可获取已经解码后的 `RpcResponse` 对象，此时我们只需将 `requestId` 与 `RpcResponse` 之间的映射关系存入 `responseMap` 即可。这样在 `RpcClient` 中就能直接获取 `RpcClientHandler` 中对 `responseMap` 对象所做的变化。进而在 `RpcClient` 中，可通过 `responseMap.get(request.getRequestId())` 来获取 `channelRead0()` 方法中传入的 `RpcResponse` 对象。与服务端请求处理器 `RpcServerHandler` 相同，此时在 `exceptionCaught()`

方法中记录异常日志，并关闭 NIO 通道即可。

当处理完 RPC 响应后，我们再来实现服务发现功能。

第三步：实现服务发现功能。

在 rpc-registry 模块中，添加一个 ServiceDiscovery 类，用于完成服务发现操作，即连接 ZooKeeper 服务器，通过服务名称来获取服务配置。

```
package demo.msa.rpc.registry;

import ...

/**
 * 服务发现
 *
 * @author huangyong
 * @since 1.0.0
 */
@Component
public class ServiceDiscovery {

    private static final Logger logger = LoggerFactory.getLogger(
(ServiceDiscovery.class));

    @Value("${rpc.registry-address}")
    private String zkAddress;

    public String discover(String name) {
        // 创建 ZooKeeper 客户端
        ZkClient zkClient = new ZkClient(zkAddress, Constant.ZK_SESSION_TIMEOUT,
Constant.ZK_CONNECTION_TIMEOUT);
        logger.debug("connect to zookeeper");
        try {
            // 获取 service 节点
            String servicePath = Constant.ZK_REGISTRY_PATH + "/" + name;
            if (!zkClient.exists(servicePath)) {
                throw new RuntimeException(String.format("can not find any service
node on path: %s", servicePath));
            }
        }
    }
}
```

```

        List<String> addressList = zkClient.getChildren(servicePath);
        if (CollectionUtil.isEmpty(addressList)) {
            throw new RuntimeException(String.format("can not find any address
node on path: %s", servicePath));
        }
        // 获取 address 节点
        String address;
        int size = addressList.size();
        if (size == 1) {
            // 若只有一个地址，则获取该地址
            address = addressList.get(0);
            logger.debug("get only address node: {}", address);
        } else {
            // 若存在多个地址，则随机获取一个地址
            address = addressList.get(ThreadLocalRandom.current().nextInt(size));
            logger.debug("get random address node: {}", address);
        }
        // 获取 address 节点的值
        String addressPath = servicePath + "/" + address;
        return zkClient.readData(addressPath);
    } finally {
        zkClient.close();
    }
}
}
}

```

我们需要从 `application.properties` 配置文件中获取 `rpc.registry-address` 配置项，该配置代表 ZooKeeper 服务注册中心的 IP 地址与端口号。在 `discover()` 方法中，首先通过 `ZkClient` 去连接 ZooKeeper 服务器。随后通过该方法的服务名称参数来拼接 `service` 节点路径（`servicePath`），并获取 `servicePath` 下的所有子节点（`address` 节点）。随后使用了一个简单的算法来获取 `address` 节点，需要考虑两种情况：若只有一个 `address` 节点时，则直接获取该节点中的数据，否则随机获取某个节点中的数据。随后通过地址节点的路径来获取服务名称所对应的服务配置。最终需要关闭与 ZooKeeper 服务器的连接，完成这次服务发现操作。

在 `rpc-hello-client` 模块中，创建一个 `application.properties` 配置文件（该文件位于 `src/main/resources` 目录下），我们在该文件中添加一个名为 `rpc.registry-address` 的配置项，值为 ZooKeeper 服务器的 IP 地址与端口号。

```
rpc.registry-address=localhost:2181
```

下面我们来完成最后一步，启动 RPC 客户端，发送 RPC 请求，以验证我们的 RPC 调用过程。

第四步：启动 RPC 客户端。

在 `rpc-hello-client` 模块中创建一个 Spring Boot 应用程序启动类 `HelloClientApplication`，在该类中注入 `RpcClient` 对象，并通过该对象来发送 RPC 请求。

```
package demo.msa.rpc.hello.client;

import ...

@SpringBootApplication(scanBasePackages = "demo.msa.rpc")
public class HelloClientApplication {

    @Autowired
    private RpcClient rpcClient;

    @PostConstruct
    public void run() {
        HelloService helloService = rpcClient.create(HelloService.class);
        System.out.println(helloService.say("world"));
    }

    public static void main(String[] args) {
        SpringApplication.run(HelloClientApplication.class, args).close();
    }
}
```

与服务端应用程序类似，我们必须指定 `scanBasePackages` 作为 Spring Boot 类扫描的基础包名，否则应用程序无法成功加载 RPC 客户端框架。在类中我们使用 `@Autowired` 注解来注入 `RpcClient` 对象。此外，我们通过定义一个带有 `@PostConstruct` 注解的方法来使用 `RpcClient` 对象，并创建 `HelloService` 接口代理对象，从而调用该对象中的目标方法。

当启动 RPC 客户端成功后，我们可在控制台中看到如下信息。

```
2017-05-23 19:54:13.618 DEBUG 4513 --- [           main]
demo.msa.rpc.registry.ServiceDiscovery : connect to zookeeper
2017-05-23 19:54:13.637 DEBUG 4513 --- [           main]
```

```
demo.msa.rpc.registry.ServiceDiscovery    : get only address node:
address-0000000000
    2017-05-23 19:54:13.667 DEBUG 4513 --- [           main]
demo.msa.rpc.client.RpcClient              : discover service:
demo.msa.rpc.hello.api.HelloService => 10.18.29.40:8000
    hello world
```

此时说明 RPC 调用成功，在控制台中输出了相关的日志，以及希望得到的 RPC 响应结果。

至此，一款轻量级分布式 RPC 框架已搭建完毕，但此时只能说完成了一个框架原型，因为还有很多地方需要改进。比如说性能方面，我们并没有做大量的压力测试，一定还存在很多需要优化的空间；再比如说扩展性方面，此框架与某些特定的技术绑定得比较紧密，若要替换底层实现，则许多地方均要做出大量调整。可见，这款 RPC 框架还存在许多不足之处，希望大家能在此基础上做出改良，让它变得更加优秀。

关于服务之间的同步调用问题，我们先探讨到这里。在下一章中，我们将话题转移到服务之间的异步调用问题上，此时客户端无须再等待服务端的响应结果就能继续完成自己接下来的任务。到底选择同步还是异步，完全根据大家的实际情况而定。

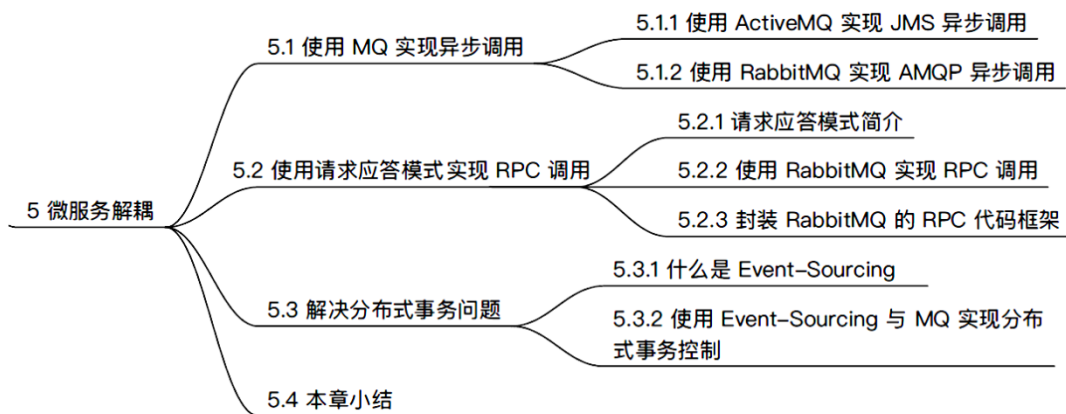
4.4 本章小结

本章围绕着微服务之间的通信来展开，目前我们仅关注同步调用问题。首先我们在 Spring Boot 应用程序中实现了基于 HTTP 的同步调用，同时我们也对比了 Spring RestTemplate、OkHttp、Retrofit 等工具的使用方法，其中每款工具都各有千秋，我们需要根据实际情况来自行选择。随后我们使用 gRPC 框架实现了基于 RPC 的同步调用，并在 Spring Boot 中整合了 gRPC，以便我们更加容易地使用 gRPC 框架来实现 RPC 同步调用。最后我们选择了亲自动手，集成 Netty、Protostuff、ZooKeeper 等开源技术，搭建了一款轻量级分布式 RPC 框架，该框架也是我们解决服务之间 RPC 调用的首选框架。

下一章我们将视角从同步转移到异步，希望能通过消息队列的方式来解决服务之间的耦合问题。

5 chapter

第 5 章 微服务解耦



5.1 使用 MQ 实现异步调用

在前面的章节中，我们探讨了服务之间的同步调用，可以使用 HTTP 或 RPC 来完成。但并非所有的调用都需要同步，有些场景下，当客户端调用服务端时，并不需要等待服务端做出响应，此时就应该使用异步调用。虽然 HTTP 与 RPC 也能稍加改造做成异步调用的方式，但实现成本相对来说比较高。更容易使用的是基于 MQ（Message Queue，消息队列）来实现服务之间的异步调用。我们现在打算研究两款强大的 MQ 技术，首先来学习老牌经典 ActiveMQ 的使用方法，并将它与 Spring Boot 应用程序集成在一起。随后我们研究相对较新的 RabbitMQ，只需稍微修改 Spring Boot 应用程序，就能将 ActiveMQ 替换为 RabbitMQ。此外，我们还会对比 ActiveMQ 与 RabbitMQ 的性能，最终决定使用其中的一种 MQ 技术，作为我们微服务架构中服务之间异步调用的“消息中心”。

下面，我们先从 ActiveMQ 开始，一起走进 MQ 的世界。

5.1.1 使用 ActiveMQ 实现 JMS 异步调用

ActiveMQ 是 Java 世界中最为流行的开源消息中间件，它不仅功能强大，而且性能稳定，始终能在功能和性能之间保持平衡。ActiveMQ 可全面支持 JMS（Java Message Service，Java 消息服务）技术规范，为 Java 应用程序提供标准的 JMS API。此外，ActiveMQ 具备与 Spring 框架进行整合的能力，曾经有很长一段时间，ActiveMQ 都是 Spring 应用程序的消息中间件标配。同样，Spring Boot 也提供了 ActiveMQ 的“开箱即用”插件，我们只需提供几项配置，就能接入 ActiveMQ，并能轻松地使用 JMS API 来编写异步消息通信程序。

关于 ActiveMQ 的更多信息，可从它的官网中加以学习，如图 5-1 所示。

ActiveMQ 官网：<http://activemq.apache.org/>。

下面我们就来一睹 ActiveMQ 的风采，体验一下它的异步调用过程，只需以下三个步骤即可。

第一步：启动 ActiveMQ 服务器。

我们可通过 Docker 容器来运行 ActiveMQ 服务器，只需通过以下 Docker 命令即可。

```
docker run \
-d \
-p 8161:8161 \
-p 61616:61616 \
-e ACTIVEMQ_ADMIN_LOGIN=admin \
-e ACTIVEMQ_ADMIN_PASSWORD=admin \
```

```
--name activemq \
webcenter/activemq
```



图 5-1 ActiveMQ 官网

目前 ActiveMQ 官方并未提供相应的 Docker 镜像,我们选择一个使用最为广泛的 ActiveMQ 第三方镜像 `webcenter/activemq`,用它来启动 ActiveMQ 容器。该镜像拥有一个基于 Web 的控制台,可通过浏览器来访问。当然,我们也可制作自己的 ActiveMQ 容器,同样也是相当容易的事情。

在启动 ActiveMQ 容器时,容器对宿主机暴露了两个端口号。

- 8186: 表示 ActiveMQ 控制台端口号,可在浏览器中通过控制台来执行 ActiveMQ 的相关操作。
- 61616: 表示 ActiveMQ 所监听的 TCP 端口号,应用程序可通过该端口号与 ActiveMQ 建立 TCP 连接,并完成后续的异步消息通信。

此外,在启动 ActiveMQ 容器时,还提供了两个环境变量。

- `ACTIVEMQ_ADMIN_LOGIN`: 用于设置控制台管理员的用户名,默认为 `admin`。
- `ACTIVEMQ_ADMIN_PASSWORD`: 用于设置控制台管理员的密码,默认为 `admin`。

关于 ActiveMQ 更为详细的使用方法,可从它的镜像站点中获知。

ActiveMQ 镜像: <https://hub.docker.com/r/webcenter/activemq/>。

ActiveMQ 容器启动完毕后，我们可打开浏览器，并在地址栏中输入 `http://localhost:8161/`，此时将看到 ActiveMQ 控制台，如图 5-2 所示。

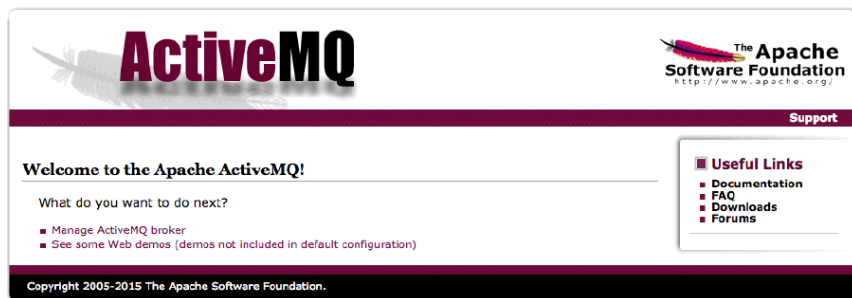


图 5-2 ActiveMQ 控制台界面

我们点击 `Manage ActiveMQ broker` 链接，浏览器将弹出一个对话框，此时浏览器要求我们输入用户名与密码用于管理员身份认证，只要浏览器不关闭，该身份认证只需一次即可。当身份认证通过后，将进入以下管理界面，如图 5-3 所示。

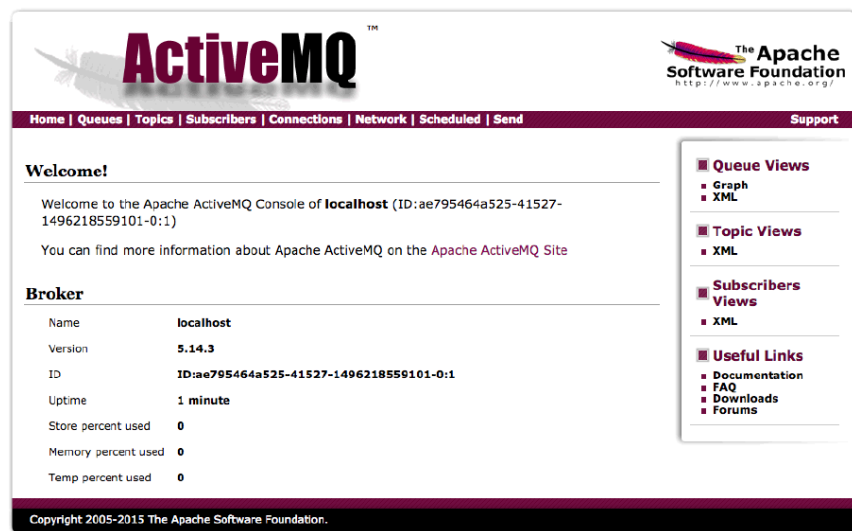


图 5-3 ActiveMQ 控制台管理界面

在以上管理界面中，包括了 8 个功能菜单。

- (1) Home: 用于查看 ActiveMQ 的基本信息。
- (2) Queues: 用于查看 ActiveMQ 所管理的队列。
- (3) Topics: 用于查看 ActiveMQ 所管理的主题。

- (4) Subscribers: 用于查看相关主题的订阅者。
- (5) Connections: 用于查看 ActiveMQ 客户端的连接信息。
- (6) Network: 用于查看 ActiveMQ 的网络相关信息。
- (7) Scheduled: 用于查看 ActiveMQ 内部运行的定时任务。
- (8) Send: 用于通过表单方式向队列或主题发送具体消息。

需要补充说明的是, ActiveMQ 管理了两类消息通道, 一类叫队列 (Queue), 另一类叫主题 (Topic)。

Queue 用于解决消息的“点对点 (Point-to-Point)”通信问题, 也就是说, 消息从生产者 (Producer) 发出后, 首先进入 ActiveMQ 某个指定的 Queue 中, 然后将此消息传送给其中一个消费者 (Consumer), 如图 5-4 所示。

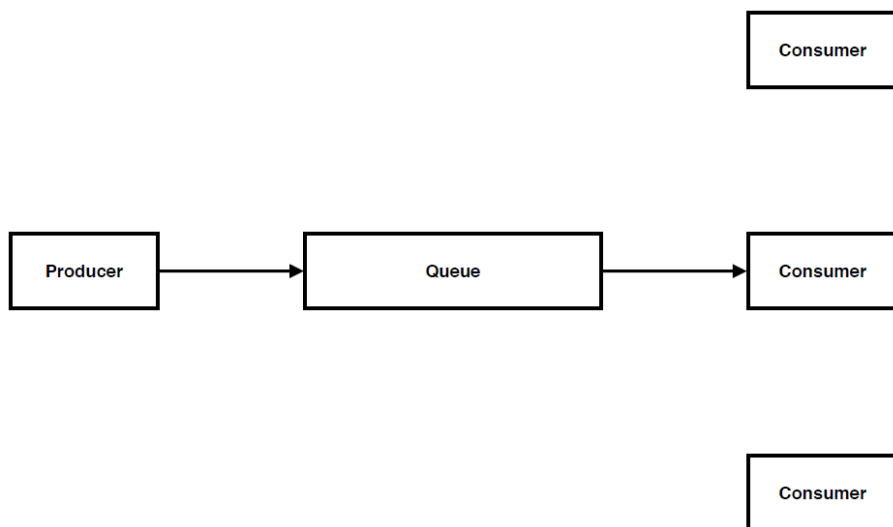


图 5-4 “点对点”消息通信模型

Topic 用于解决消息的“发布与订阅 (Publish-Subscribe)”通信问题, 也就是说, 消息从 Producer 发出后, 首先将其发布到 ActiveMQ 某个指定的 Topic 上, 然后将此消息分发给每个订阅者 (Subscriber), 如图 5-5 所示。

可在具体的场合下, 灵活使用以上两种通信模式来实现 Producer 与 Consumer/Subscriber 之间的异步调用, 从而解决调用方与被调用方的耦合问题。可见, Queue 能解决调用缓冲问题, Topic 能解决消息广播问题, Queue 与 Topic 都能解决调用耦合问题, 这些技术都为一个好的软件架构提供了有效的支撑。

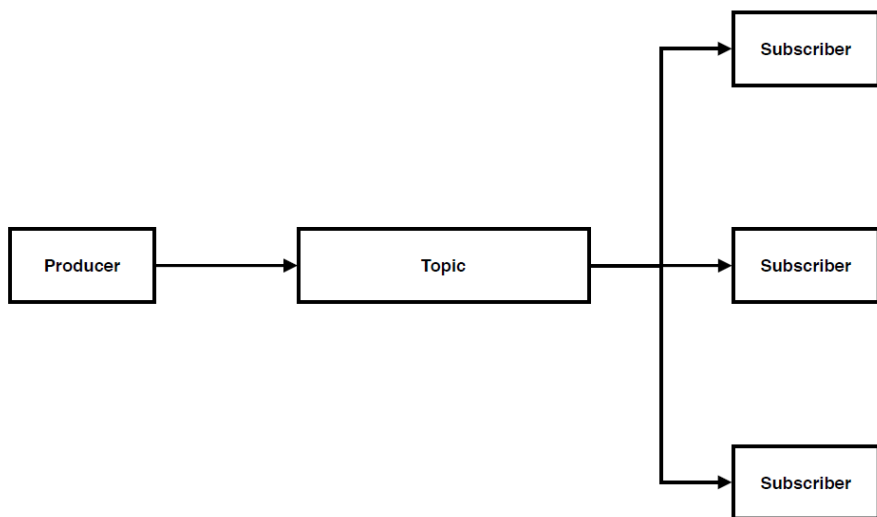


图 5-5 “订阅与发布”消息通信模型

下面就以 Queue 为例，将 ActiveMQ 与 Spring Boot 进行整合，将 Producer 作为客户端，将 Consumer 作为服务端，通过 Queue 来实现客户端与服务端的异步调用，现在我们就从服务端开始。

第二步：开发服务端（消费者）。

创建一个名为 activemq-hello-server 的 Maven 项目，在 pom.xml 配置文件中添加以下 Maven 依赖。

```
...  
<parent>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-parent</artifactId>  
  <version>1.5.2.RELEASE</version>  
</parent>  
  
<dependencies>  
  <dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-activemq</artifactId>  
  </dependency>  
</dependencies>  
...
```

在 Spring Boot 框架中已经内置了对 ActiveMQ 的支持，我们只需依赖 `spring-boot-starter-activemq` 就能启用 ActiveMQ，此时还需在 `application.properties` 配置文件中添加 ActiveMQ 的相关配置项。

```
spring.activemq.broker-url=tcp://localhost:61616
spring.activemq.user=admin
spring.activemq.password=admin
```

第一项配置表示 ActiveMQ 的连接 URL 地址，包括协议名、主机名与端口号，后两项配置表示登录 ActiveMQ 的用户名与密码。

接下来创建一个名为 `HelloServer` 的类，封装服务端相关代码。

```
package demo.msa.activemq.hello.server;

import org.springframework.jms.annotation.JmsListener;
import org.springframework.stereotype.Component;

@Component
public class HelloServer {

    @JmsListener(destination = "hello-queue")
    public void receive(String message) {
        System.out.println(message);
    }
}
```

该类带有 `@Component` 注解，说明它可被 Spring IoC 容器所管理。此时只需使用 Spring JMS 所提供的 `@JmsListener` 注解，并将其绑定到 `receive()` 方法上，就能从 ActiveMQ 中接收相应的消息，该方法名可以任意指定。在 `@JmsListener` 注解中需添加一个 `destination` 属性来指定 Queue/Topic 的名称，该名称需要具备唯一性。消息将以一个 `String` 类型参数的形式传入方法体中，也可接收其他类型的消息，这取决于客户端发送的消息是哪种类型。Spring JMS 将消息放入 ActiveMQ 时会进行序列化，当消息从 ActiveMQ 取出时将进行反序列化，应用程序无须关注这些底层细节，我们只需将精力集中在业务逻辑上。

最后，我们编写一个 Spring Boot 应用程序启动类来启动服务端。

```
package demo.msa.activemq.hello.server;
```

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class HelloServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(HelloServerApplication.class, args);
    }
}
```

当服务端启动完毕后，将持续监听 ActiveMQ 的 hello-queue 队列中即将到来的消息，该消息由客户端来发送。

第三步：开发客户端（生产者）。

创建一个名为 activemq-hello-client 的 Maven 项目，pom.xml 配置文件中的内容与 activemq-hello-server 项目相似。

接下来创建一个名为 HelloClient 的类，将其作为客户端。

```
package demo.msa.activemq.hello.client;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Component;

@Component
public class HelloClient {

    @Autowired
    private JmsTemplate jmsTemplate;

    public void send(String message) {
        jmsTemplate.convertAndSend("hello-queue", message);
    }
}
```

该类同样带有 @Component 注解，与 HelloServer 不同的是，我们在这里使用了 @Autowired

注解，将 `JmsTemplate` 对象注入进来，还编写了一个 `send()` 方法，在该方法中调用 `JmsTemplate` 对象的 `convertAndSend()` 方法来转换并发送消息（转换指的是消息的序列化），所调用的方法中需指定对应的队列名称，该队列名称与 `HelloServer` 中所监听的队列名称保持一致。

客户端也需要提供 `application.properties` 配置文件，其文件内容与 `activemq-hello-server` 项目完全一致。

最后我们编写一个 `Spring Boot` 应用程序启动类来启动客户端。

```
import org.springframework.boot.autoconfigure.SpringBootApplication;

import javax.annotation.PostConstruct;

@SpringBootApplication
public class HelloClientApplication {

    @Autowired
    private HelloClient helloClient;

    @PostConstruct
    public void init() {
        helloClient.send("hello world");
    }

    public static void main(String[] args) {
        SpringApplication.run(HelloClientApplication.class, args).close();
    }
}
```

此时，我们通过 `@Autowired` 注解来注入 `HelloClient` 对象，并在 `init()` 方法中调用 `HelloClient` 对象的 `send()` 方法来发送具体的消息内容。需要注意的是，`init()` 方法带有 `@PostConstruct` 注解，表示 `Spring IoC` 容器实例化 `HelloClientApplication` 类后将调用该方法，确切地说是在调用构造方法以后，`Spring` 框架将自动调用带有 `@PostConstruct` 注解的 `init()` 方法。`Spring IoC` 容器的生命周期从调用 `main()` 方法中的 `SpringApplication.run()` 方法开始，到调用 `close()` 方法而结束。

运行 `main()` 方法可启动客户端应用程序，此时可以在服务端应用程序控制台中看到客户端所发送的消息。当然，也可在 `ActiveMQ` 控制台中查看队列的当前状态，如图 5-6 所示。

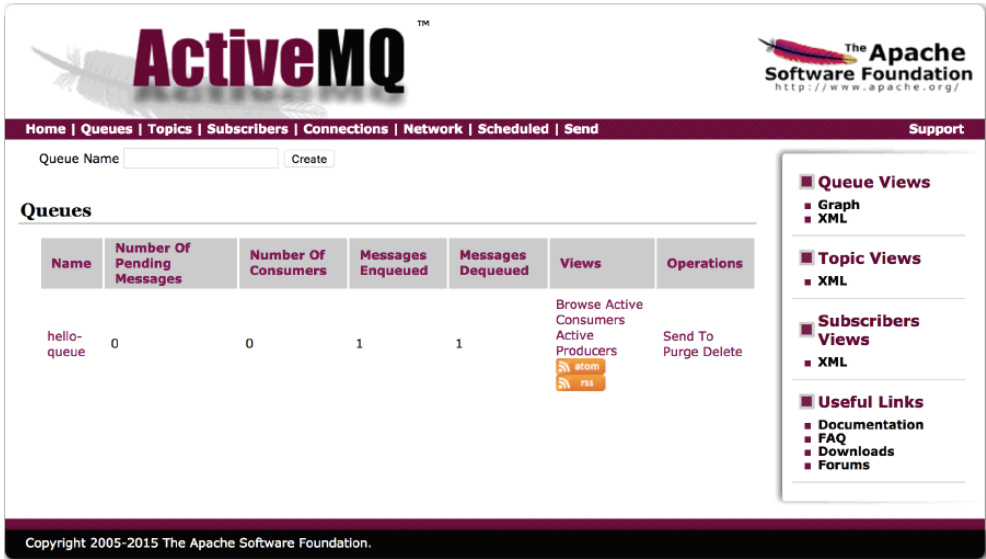


图 5-6 ActiveMQ 队列当前状态

在 Queues 表格中，出现了一些特殊含义的列名，下面对其分别进行描述。

- **Name:** 表示队列名称，可在应用程序中自动创建，也可在 ActiveMQ 控制台中手工创建。
- **Number Of Pending Messages:** 表示阻塞在队列中未经消费的消息条数。
- **Number Of Consumers:** 表示正在与 ActiveMQ 建立连接的消费者数量。
- **Messages Enqueued:** 表示进入队列的消息数量。
- **Messages Dequeued:** 表示离开队列的消息数量。

此外，这里还有几种操作，也有必要解释一下。

- **Browse:** 用于查看当前队列中消息的相关细节，也可点击 Name 列中的链接来执行同样的操作。
- **Active Consumers:** 用于查看当前活动消费者的相关信息。
- **Active Producers:** 用于查看当前活动生产者的相关信息。
- **Send To:** 用于向当前队列中发送具体消息。
- **Purge:** 用于清空队列中的消息。
- **Delete:** 用于删除当前队列。

至此，客户端 `activemq-hello-client` 已成功将消息通过 ActiveMQ 发送至服务端 `activemq-hello-server`，此通信过程是异步的，客户端无须等待服务端响应。

那么 ActiveMQ 的性能如何呢？我们不妨做一个简单的性能测试，从客户端循环发送 1000 条消息，统计需要耗用多少时间。

```
package demo.msa.activemq.hello.client;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.util.StopWatch;

import javax.annotation.PostConstruct;

@SpringBootApplication
public class HelloClientApplication {

    @Autowired
    private HelloClient helloClient;

    @PostConstruct
    public void init() {
        StopWatch stopWatch = new StopWatch();
        stopWatch.start();

        for (int i = 0; i < 1000; i++) {
            helloClient.send("hello world");
        }

        stopWatch.stop();
        System.out.println("time: " + stopWatch.getTotalTimeMillis() + "ms");
    }

    public static void main(String[] args) {
        SpringApplication.run(HelloClientApplication.class, args).close();
    }
}
```

我们利用 Spring 框架自带的 StopWatch 工具类来实现计时器功能，在循环开始之前调用 StopWatch 对象的 start()方法来开始计时，在循环结束之后调用 StopWatch 对象的 stop()方法来

停止计时，最后通过调用 `StopWatch` 对象的 `getTotalTimeMillis()` 方法来获取计时器所记录的总时长，这是一个毫秒数，我们将其打印到控制台。

运行 `HelloClientApplication` 程序，在控制台输出：`time: 56414ms`。可见，客户端循环发送 1000 条消息几乎需要 1 分钟时间，`ActiveMQ` 的性能看起来并不是特别高。

当程序中有大量的消息需要从客户端发送至服务端，而且需要确保消息发送过程具备较高的性能，此时有更好的技术方案可以解决吗？

我们得知业界还有一种流行的开源消息中间件叫 `RabbitMQ`，听说它可具备更高的性能，是否真的如此呢？我们接下来就用 `RabbitMQ` 实现以上相同的异步调用过程，并与 `ActiveMQ` 进行非常直观的性能对比。

5.1.2 使用 RabbitMQ 实现 AMQP 异步调用

`RabbitMQ` 不再基于 `JMS` 规范，也没选用 `Java` 语言作为底层技术，而是基于另一种消息通信协议，名为 `AMQP`（`Advanced Message Queuing Protocol`，高级消息队列协议），并采用性能更加突出的 `Erlang` 语言作为技术实现。`RabbitMQ` 提供了众多编程语言客户端（包括官方和第三方），当然也能与 `Spring` 框架进行整合，`Spring Boot` 同样也提供了对 `RabbitMQ` 的支持。

关于 `RabbitMQ` 的更多信息，可从它的官网中加以学习，如图 5-7 所示。



图 5-7 RabbitMQ 官网

`RabbitMQ` 官网：<http://www.rabbitmq.com/>。

从 RabbitMQ 的官网中可知，它的功能比 ActiveMQ 更加强大和灵活，官方教程也比 ActiveMQ 更加全面和专业。通过学习官方教程，我们可快速了解 RabbitMQ 的原理以及使用方法。下面我们仍然通过三个步骤来体验 RabbitMQ，实现客户端（生产者）与服务端（消费者）的异步调用过程。

第一步：启动 RabbitMQ 服务器。

RabbitMQ 官方已经提供了自己的 Docker 容器，我们可使用以下 Docker 命令来启动 RabbitMQ 服务器。

```
docker run \
-d \
-p 15672:15672 \
-p 5672:5672 \
-e RABBITMQ_DEFAULT_USER=admin \
-e RABBITMQ_DEFAULT_PASS=admin \
--name rabbitmq \
rabbitmq:3-management
```

此时，我们使用了 `rabbitmq:3-management` 镜像启动 RabbitMQ 容器。该镜像拥有一个基于 Web 的控制台，可通过浏览器来访问，这一点与 ActiveMQ 类似。RabbitMQ 除了控制台，还提供了 HTTP API 与命令客户端等方式，前者可用于应用程序使用，后者方便运维人员使用。如果不想启动控制台 Web 应用程序，那么可以选择 `rabbitmq:3` 镜像来启动 RabbitMQ 容器。

在启动 RabbitMQ 容器时，容器对宿主机暴露了两个端口号。

- 15672：表示 RabbitMQ 控制台端口号，可在浏览器中通过控制台来执行 RabbitMQ 的相关操作。
- 5672：表示 RabbitMQ 所监听的 TCP 端口号，应用程序可通过该端口号与 RabbitMQ 建立 TCP 连接，并完成后续的异步消息通信。

此外，在启动 RabbitMQ 容器时，还提供了两个环境变量。

- `RABBITMQ_DEFAULT_USER`：用于设置 RabbitMQ 控制台默认用户的用户名，默认为 `guest`。
- `RABBITMQ_DEFAULT_PASS`：用于设置 RabbitMQ 控制台默认用户的密码，默认为 `guest`。

关于 RabbitMQ 更为详细的使用方法，可从它的镜像站点中获知。

RabbitMQ 镜像：https://hub.docker.com/_/rabbitmq/。

RabbitMQ 容器启动完毕后，我们可打开浏览器，并在地址栏中输入 `http://localhost:15672/`，此时需输入登录用户的用户名与密码，随后将看到 RabbitMQ 控制台，如图 5-8 所示。

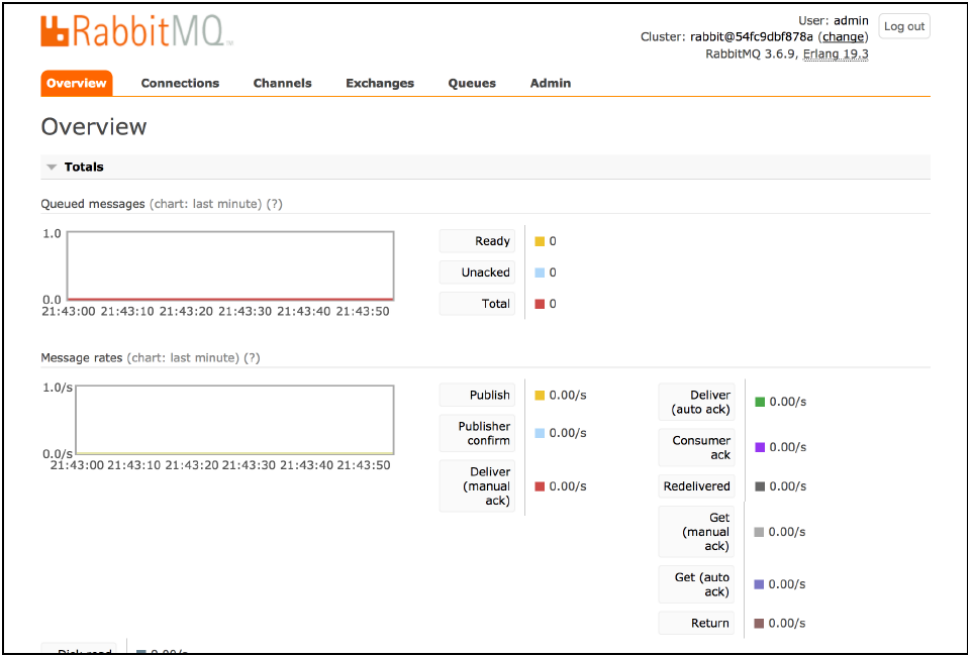


图 5-8 RabbitMQ 控制台

在以上管理界面中，包括了 6 个功能菜单。

- (1) Overview: 用于查看 RabbitMQ 的基本信息。
- (2) Connections: 用于查看 RabbitMQ 客户端的连接信息。
- (3) Channels: 用于查看 RabbitMQ 的通道。
- (4) Exchanges: 用于查看 RabbitMQ 的交换机。
- (5) Queues: 用于查看 RabbitMQ 的队列。
- (6) Admin: 用于管理 RabbitMQ 的用户、虚拟主机、策略等数据。

需要补充说明的是，RabbitMQ 只有 Queue，没有 Topic，因为可通过 Exchange 与 Queue 的组合来实现 Topic 所具备的功能，这样的结构更加简单和灵活，以下便是 RabbitMQ/AMQP 的消息模型，如图 5-9 所示。

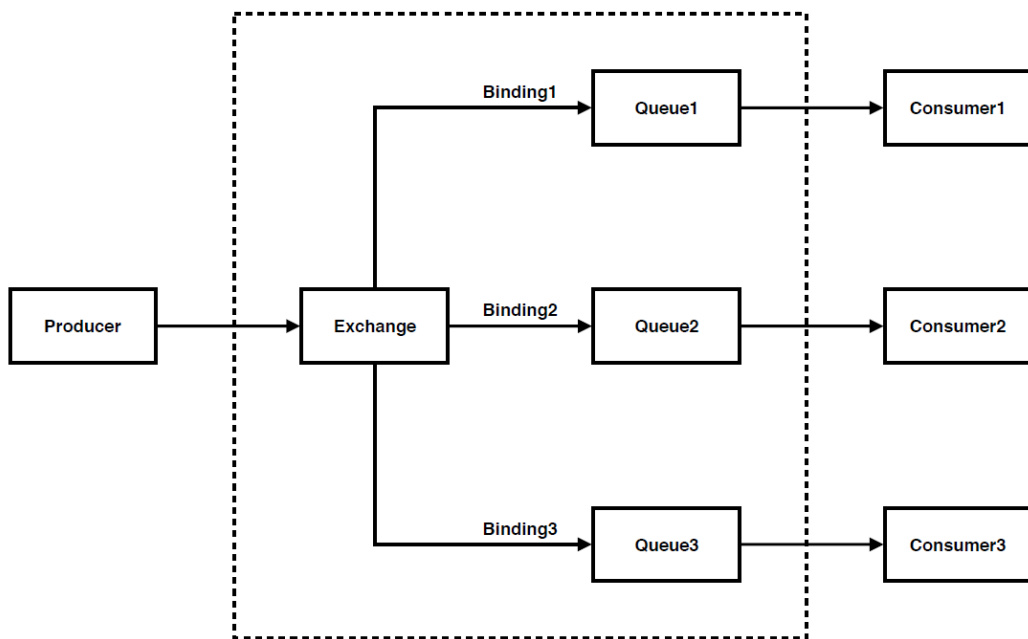


图 5-9 RabbitMQ/AMQP 消息通信模型

在 Exchange 与 Queue 之间有一个 Binding 关系,当消息从 Producer 发送至 Exchange 中时,会根据 Binding 来路由消息的去向。如果 Binding 各不相同,那么该消息将路由到其中一个 Queue 中,随后将被一个 Consumer 所消费,此时实现了“点对点”消息通信模型;如果 Binding 完全相同,那么该消息将路由到每个 Queue 中,随后将被每个 Consumer 所消费,此时实现了“发布与订阅”消息通信模型。因此,可将 Binding 理解为 Exchange 到 Queue 的路由规则,这些规则可通过 RabbitMQ 所提供的客户端 API 来控制,也可通过 RabbitMQ 所提供控制台来管理。

需要补充说明的是, RabbitMQ 提供了一个默认的 Exchange (名称为 AMQP default),在 RabbitMQ 控制台的 Exchanges 菜单中就能看到它。简单情况下,我们只需使用默认的 Exchange 即可,当需要提供发布与订阅功能时才会使用自定义的 Exchange。

下面我们就将 Spring Boot 与 RabbitMQ 进行整合。先开发一个服务端作为消息的消费者,再开发一个客户端作为消息的生产者,随后运行客户端,并查看服务端中接收到的消息,从而实现客户端与服务端之间的异步通信过程。

第二步:开发服务端(消费者)。

创建一个名为 rabbitmq-hello-server 的 Maven 项目,在 pom.xml 配置文件中添加以下 Maven 依赖。

```

...
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.5.2.RELEASE</version>
</parent>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
  </dependency>
</dependencies>
...

```

在 Spring Boot 框架中已经内置了对 RabbitMQ 的支持，我们只需依赖 `spring-boot-starter-amqp` 就能启用 RabbitMQ，此时还需在 `application.properties` 配置文件中添加 RabbitMQ 的相关配置项。

```

spring.rabbitmq.addresses=localhost:5672
spring.rabbitmq.username=admin
spring.rabbitmq.password=admin

```

第一项配置表示 RabbitMQ 的连接主机名与端口号，后两项配置表示登录 RabbitMQ 的用户名与密码。

接下来创建一个名为 `HelloServer` 的类，封装服务端相关代码。

```

package demo.msa.rabbitmq.hello.server;

import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;

@Component
public class HelloServer {

    @RabbitListener(queues = "hello-queue")
    public void receive(String message) {
        System.out.println(message);
    }
}

```



```
    }  
}
```

以上代码与 `activemq-hello-server` 项目中的 `HelloServer` 代码类似，只是在 `receive()` 方法上定义了 `@RabbitListener`，需要在该注解中设置 `queues` 参数来指定消费者需要监听的队列名称。

最后我们编写一个 `Spring Boot` 应用程序启动类来启动服务端。

```
package demo.msa.rabbitmq.hello.server;  
  
import org.springframework.amqp.core.Queue;  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.context.annotation.Bean;  
  
@SpringBootApplication  
public class HelloServerApplication {  
  
    @Bean  
    public Queue helloQueue() {  
        return new Queue("hello-queue");  
    }  
  
    public static void main(String[] args) {  
        SpringApplication.run(HelloServerApplication.class, args);  
    }  
}
```

以上代码与 `activemq-hello-server` 项目中的 `HelloServerApplication` 代码类似，只是这里使用了一个带有 `@Bean` 注解的 `helloQueue()` 方法创建了一个名为 `hello-queue` 的队列。在 `RabbitMQ` 中，必须通过程序来创建队列，不像 `RabbitMQ` 那样，在用到队列的时候来动态创建。

当服务端启动完毕后，将持续监听 `RabbitMQ` 的 `hello-queue` 队列中即将到来的消息，该消息由客户端来发送。

第三步：开发客户端（生产者）。

创建一个名为 `rabbitmq-hello-client` 的 `Maven` 项目，`pom.xml` 配置文件中的内容与 `rabbitmq-hello-server` 项目相似。

接下来创建一个名为 `HelloClient` 的类，将其作为客户端。

```
package demo.msa.rabbitmq.hello.client;

import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class HelloClient {

    @Autowired
    private RabbitTemplate rabbitTemplate;

    public void send(String message) {
        rabbitTemplate.convertAndSend("hello-queue", message);
    }
}
```

以上代码与 `activemq-hello-client` 项目中的 `HelloClient` 代码类似，只是这里使用 `@Autowired` 注解注入了 `RabbitTemplate` 对象，并调用该对象的 `convertAndSend()` 方法，向指定的队列中发送消息。

客户端也需要提供 `application.properties` 配置文件，其文件内容与 `rabbitmq-hello-server` 项目完全一致。

最后我们编写一个 `Spring Boot` 应用程序启动类来启动客户端。

```
package demo.msa.rabbitmq.hello.client;

import org.springframework.amqp.core.Queue;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

import javax.annotation.PostConstruct;

@SpringBootApplication
```

```
public class HelloClientApplication {

    @Autowired
    private HelloClient helloClient;

    @Bean
    public Queue helloQueue() {
        return new Queue("hello-queue");
    }

    @PostConstruct
    public void init() {
        helloClient.send("hello world");
    }

    public static void main(String[] args) {
        SpringApplication.run(HelloClientApplication.class, args).close();
    }
}
```

以上代码与 `activemq-hello-client` 项目中的 `HelloClientApplication` 代码类似，只是这里同样也使用一个带有 `@Bean` 注解的 `helloQueue()` 方法创建了一个名为 `hello-queue` 的队列，这样做的好处是，可以确保当客户端在服务端之前启动时，也能创建所需的队列。随后当服务端启动时，不会出现找不到队列的错误。此外，由于 `RabbitMQ` 可以确保不会创建同名的队列，因此我们可分别在服务端与客户端中创建同名的队列。

运行 `main()` 方法可启动客户端应用程序，此时将看到服务端应用程序控制台中看到客户端所发送的消息。当然，也可在 `RabbitMQ` 控制台中查看消息队列当前状态，如图 5-10 所示。

至此，客户端 `rabbitmq-hello-client` 已成功将消息通过 `RabbitMQ` 发送至服务端 `rabbitmq-hello-server`，此通信过程是异步的，客户端无须等待服务端响应。



图 5-10 RabbitMQ 队列当前状态

需要补充说明的是，目前我们发送的消息是 `String` 类型，这是 JDK 内置的包装类型，如果需要发送的消息是一个普通的 `Java Bean` 类型，是否也能正常调用呢？经验证，`Java Bean` 必须实现 `java.io.Serializable` 序列化接口才能成功完成调用。原因很简单，因为 RabbitMQ 所传送的消息是 `byte[]` 类型，当客户端发送消息后需要进行序列化（将 `Java` 类型转为 `byte[]` 类型），当服务端接收消息前需要进行反序列化（将 `byte[]` 类型转为 `Java` 类型）。然而，RabbitMQ 默认使用的是 JDK 自带的序列化方式，因此我们发送的消息对象必须实现 JDK 的序列化接口。当然，我们也想使用其他更加高效的序列化方式，比如使用 Jackson 来实现 `Java` 对象与 `JSON` 字符串之间的序列化与反序列化操作。幸运的是，RabbitMQ 已经我们提供了 Jackson 序列化的方式，我们只需在服务端与客户端的 Spring Boot 应用程序启动类中，定义一个 `Jackson2JsonMessageConverter` 的 Spring Bean 即可。

```
@Bean
public Jackson2JsonMessageConverter messageConverter() {
    return new Jackson2JsonMessageConverter();
}
```

大家不妨尝试使用以上方法来发送 **Java Bean** 消息，观察调用是否成功。

那么 **RabbitMQ** 的性能如何呢？我们做一个与 **ActiveMQ** 一样的性能测试，从客户端循环发送 1000 条消息，统计需要耗用多少时间。

```
package demo.msa.rabbitmq.hello.client;

import org.springframework.amqp.core.Queue;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.util.StopWatch;

import javax.annotation.PostConstruct;

@SpringBootApplication
public class HelloClientApplication {

    @Autowired
    private HelloClient helloClient;

    @Bean
    public Queue helloQueue() {
        return new Queue("hello-queue");
    }

    @PostConstruct
    public void init() {
        StopWatch stopWatch = new StopWatch();
        stopWatch.start();

        for (int i = 0; i < 1000; i++) {
            helloClient.send("hello world");
        }

        stopWatch.stop();
        System.out.println("time: " + stopWatch.getTotalTimeMillis() + "ms");
    }
}
```

```
public static void main(String[] args) {  
    SpringApplication.run(HelloClientApplication.class, args).close();  
}  
}
```

运行 `HelloClientApplication` 程序，在控制台中输出：`time: 356ms`。可见，客户端循环发送 1000 条消息只需不足 0.5 秒的时间，做同样的操作 `ActiveMQ` 却需要 1 分钟的时间，差距是 158 倍。然而以上只是针对单线程的情况，在多线程并发的情况下，`RabbitMQ` 的性能是否能保持稳定呢？我们继续进行下面的性能测试。

我们创建了 10 个线程，每个线程并发地向 `RabbitMQ` 中写入 1000 条消息（一共写入 10000 条消息），运行以下程序并观察所耗时间。

```
package demo.msa.rabbitmq.hello.client;  
  
import org.springframework.amqp.core.Queue;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.context.annotation.Bean;  
import org.springframework.util.StopWatch;  
  
import javax.annotation.PostConstruct;  
import java.util.concurrent.CountDownLatch;  
import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
  
@SpringBootApplication  
public class HelloClientApplication {  
  
    @Autowired  
    private HelloClient helloClient;  
  
    @Bean  
    public Queue helloQueue() {  
        return new Queue("hello-queue");  
    }  
}
```

```
@PostConstruct
public void init() {
    Stopwatch watch = new Stopwatch();
    watch.start();

    int threads = 10;
    ExecutorService pool = Executors.newFixedThreadPool(threads);
    try {
        final CountDownLatch begin = new CountDownLatch(1);
        final CountDownLatch end = new CountDownLatch(threads);
        for (int n = 0; n < threads; n++) {
            pool.execute(new Runnable() {
                @Override
                public void run() {
                    try {
                        begin.await();
                        for (int i = 0; i < 1000; i++) {
                            helloClient.send("hello world");
                        }
                    } catch (Exception e) {
                        e.printStackTrace();
                    } finally {
                        end.countDown();
                    }
                }
            });
        }
        begin.countDown();
        end.await();
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        pool.shutdown();
    }

    watch.stop();
    System.out.println("time: " + watch.getTotalTimeMillis() + "ms");
}
```

```
public static void main(String[] args) {  
    SpringApplication.run(HelloClientApplication.class, args).close();  
}  
}
```

我们使用了 JDK 并发包中的 `ExecutorService` 与 `CountDownLatch` 来执行多线程的并发调用过程。使用 `ExecutorService` 创建了一个固定大小的线程池，该线程池的大小正好容纳所有的线程数，使用线程池是为了避免创建和销毁线程时所产生的性能开销。使用了两个 `CountDownLatch`（`begin` 与 `end`）来控制多线程的并发执行，`begin` 用于确保子线程能在同一时刻并发执行，`end` 用于确保所有子线程执行完毕后才继续执行主线程。当调用 `CountDownLatch` 对象的 `await()` 方法时，当前线程将在此处处于等待状态；当调用 `CountDownLatch` 对象 `countDown()` 方法时，会将 `CountDownLatch` 内部的计数器自动减 1，当计数器为 0 时，`await()` 方法执行完毕，并执行接下来的程序。

运行 `HelloClientApplication` 程序，在控制台中输出：`time: 541ms`。说明 10 个并发对 RabbitMQ 的吞吐量并没有明显降低，性能依然稳定。此时，在服务端控制台中可看到输出的 10000 条消息。也能在 RabbitMQ 控制台中看到 `hello-queue` 队列在一段时间内的状态变化，如图 5-11 所示。



图 5-11 RabbitMQ 队列状态变化

可见, RabbitMQ 的性能比 ActiveMQ 更加高效和稳定, 同样也能非常方便地与 Spring Boot 应用程序集成, 还拥有更加丰富的官方文档, 以及更加强大的控制台。因此, 我们决定使用放弃 ActiveMQ 而选用 RabbitMQ 作为服务之间的异步消息调用平台, 它将成为整个微服务架构中异步调用的“消息中心”。

5.2 使用请求应答模式实现 RPC 调用

当两个服务之间需要进行通信时, 我们可利用消息队列来解决此问题, 它不仅能解除服务之间的耦合性, 还能为通信过程提供缓冲, 提高系统的吞吐率。但要做到这一切, 我们需要接受异步调用这种通信方式, 如果在实际情况下, 要求服务之间的调用必须做到同步, 此时消息队列似乎就会遇到麻烦。难道鱼和熊掌就真的不能兼得吗? 听说有“请求应答模式”可解决消息队列的同步调用问题, 而且基于 JMS 规范的 ActiveMQ 与基于 AMQP 协议的 RabbitMQ 都提供了请求应答模式的技术实现, 那么具体应该如何使用呢? 既然是同步调用, 那么 MQ 的同步与传统的 RPC 同步有何区别呢? 我们是否应该使用 MQ 来开发一个基于请求应答模式的 RPC 调用框架呢?

带着这些问题, 我们开始探险, 先从请求应答模式起航。

5.2.1 请求应答模式简介

请求应答模式 (Request-Reply) 是一种经典的企业集成模式 (Enterprise Integration Patterns), 用于在两个应用之间进行消息通信, 此时的消息不再是从生产者到消费者这样的单向流动, 而是一个“消息闭环”, 消息的流动包括以下四个步骤。

- (1) 请求者发送请求消息至请求队列中。
- (2) 应答者从请求队列中获取请求消息。
- (3) 消费者发送应答消息至应答队列中。
- (4) 请求者从应答队列中获取应答消息。

可在企业集成模式的官网中, 进一步了解请求应答模式模式。

请求应答模式: <http://www.enterpriseintegrationpatterns.com/patterns/messaging/RequestReply.html>。

请求应答模式官方提供了一幅非常清晰的架构设计图, 如图 5-12 所示。

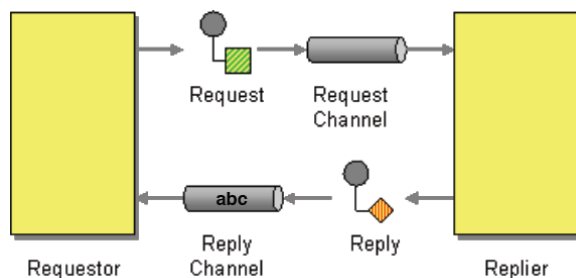


图 5-12 请求应答模式架构设计图

为了便于描述，我们有必要对图 5-12 中的核心成员进行说明。

- **Requestor**: 表示消息的请求者，消息从这里发出，也回到这里完成闭环。
- **Request**: 表示由请求者生产的请求消息，其中封装了请求参数。
- **Request Channel**: 表示存放请求消息的请求通道，或称为请求队列。
- **Replier**: 表示消息的应答者，用于获取请求消息，完成业务逻辑，并发送应答消息。
- **Reply**: 表示由应答者生产的应答消息，其中封装了响应结果。
- **Reply Channel**: 表示存放应答消息的请求通道，或称为应答队列。

可见，在请求应答模式中，请求者不是单纯的生产者，应答者也不是单纯的消费者。实际上，在消息的请求过程中，请求者扮演了生产者角色，应答者扮演了消费者角色。但是，在消息的应答过程中，它们的角色发生了互换，以前的响应者扮演了生产者角色，以前的请求者扮演了消费者角色。

既然我们已经选择了 RabbitMQ 作为消息中间件，那么如何使用 RabbitMQ 来实现请求应答模式呢？这是我们接下来要解决的问题。

通过学习得知，在 RabbitMQ 中，每个消息分为三个部分，第一部分叫“消息头部（Message Headers）”，第二部分叫“消息属性（Message Properties）”，最后一部分叫“消息载荷（Message Payload）”。其中，消息载荷比较好理解，它就是我们z需要发送的消息内容，在消息内部是编码（序列化）后的数据。消息头部和消息属性的结构比较类似，它们都是“键值对”结构，但在使用上也有区别。我们可以定义任意的“键值对”作为消息头部，但是消息属性却只有以下 12 种。

- （1）**content_type**: 用于设置消息的内容类型，也就是消息编码的 mime-type，例如，我们一般会将 JSON 格式消息的 content_type 属性设置为 application/json。
- （2）**content_encoding**: 用于设置消息的内容编码，默认情况下一般都是 UTF-8。
- （3）**priority**: 用于设置消息的优先级。

- (4) `correlation_id`: 用于设置消息的关联 ID, 可用于请求应答模式。
- (5) `reply_to`: 用于设置消息的应答队列名称, 可用于请求应答模式。
- (6) `expiration`: 用于设置消息的过期时间。
- (7) `message_id`: 用于设置消息的唯一性 ID。
- (8) `timestamp`: 用于设置消息的创建时间戳。
- (9) `type`: 用于设置消息的类型。
- (10) `user_id`: 用于设置消息所关联的用户 ID。
- (11) `app_id`: 用于设置消息所关联的应用 ID。
- (12) `cluster_id`: 用于设置消息所在的 RabbitMQ 集群 ID。

可见, 我们可利用 RabbitMQ 中 `correlation_id` 与 `reply_to` 这两个消息属性来实现请求应答模式。

结合上面的请求应答模式架构设计图与 RabbitMQ 的消息属性, 我们做出以下设计, 如图 5-13 所示。

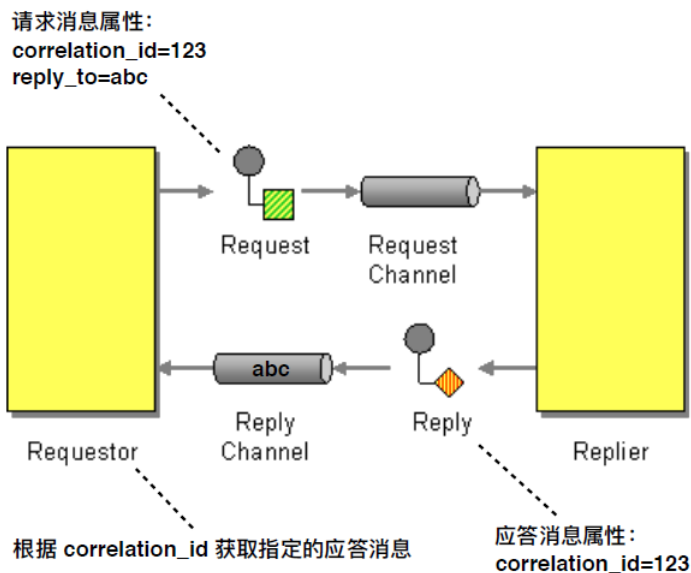


图 5-13 通过消息属性实现请求应答模式

当请求消息从请求者发出后, 该消息带有两个属性: `correlation_id` 是消息的关联 ID, 可通过一个随机数、时间戳、UUID 等方式来自动生成, 只需保证唯一性即可; `reply_to` 是应答队列的名称, 也可随机生成。

当应答消息从应答者发出后，该消息只带有一个属性：`correlation_id`，它来自请求消息中的 `correlation_id`，也就是说，只有使应答消息的 `correlation_id` 与请求消息的 `correlation_id` 保持一致，才能在请求者中根据 `correlation_id` 来获取指定的应答消息。

一般情况下，请求者与应答者都是通过远程方式进行通信的，也就是说，请求应答模式本质上也是一种 RPC 调用，下面我们就使用 RabbitMQ 来实现这个 RPC 调用过程。

5.2.2 使用 RabbitMQ 实现 RPC 调用

在 RPC 调用中，有客户端与服务端两种角色，客户端相当于请求应答模式中的请求者，服务端相当于请求应答模式中的应答者。客户端将请求消息发送至 RabbitMQ，随后可从 RabbitMQ 中获取应答消息，这同样是一个同步的 RPC 调用过程。换句话说，消息队列也并非只能做异步调用，对于同步调用是可以通过请求应答模式来实现的。

实际上，使用 RabbitMQ 就能将请求应答模式可以进行简化，如图 5-14 所示。

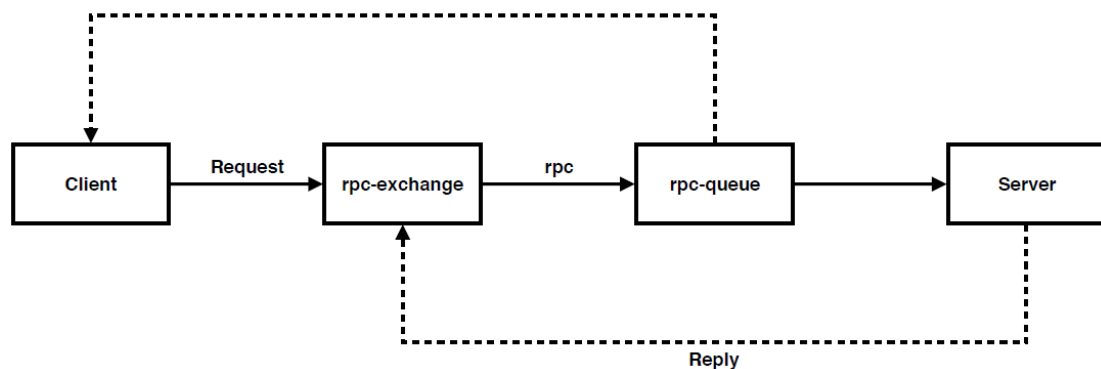


图 5-14 RabbitMQ 对请求应答模式的简化

客户端（Client）发送请求消息（Request）到 RabbitMQ 的 Exchange（`rpc-exchange`）中，随后通过 RoutingKey（`rpc`）将消息转发到 Queue（`rpc-queue`）中，最后服务端（Server）从 Queue 中获取请求消息。当 Server 处理完自己的业务逻辑后，将发送应答消息（Reply）到同样的 Exchange 中，随后通过同样的 Binding 将消息转发到同样的 Queue 中。可见，请求消息从 Client 到 Exchange，再到 Queue，最后到 Server；应答消息从 Server 到 Exchange，再到 Queue，最后到 Client。请求消息与应答消息经历了同样的 Exchange 与 Queue，去程与返程都走了同样的路，无须将应答消息放入另一个 Queue 中，从而降低了程序的复杂度，这一切得益于 RabbitMQ 的灵活性与高性能所提供的帮助。

下面，我们将利用 RabbitMQ 与 Spring Boot 实现 RPC 调用，仍然先从 RPC 服务端开始。

以下是 RPC 服务端 Spring Boot 应用程序的启动类代码，我们需要服务端所需的 Queue 与 Exchange，以及它们之间的 Binding 关系。

```
package demo.msa.rabbitmq.hello.server;

import org.springframework.amqp.core.Binding;
import org.springframework.amqp.core.BindingBuilder;
import org.springframework.amqp.core.DirectExchange;
import org.springframework.amqp.core.Queue;
import org.springframework.amqp.support.converter.Jackson2JsonMessageConverter;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class HelloServerApplication {

    @Bean
    public Queue queue() {
        return new Queue("rpc-queue");
    }

    @Bean
    public DirectExchange exchange() {
        return new DirectExchange("rpc-exchange");
    }

    @Bean
    public Binding binding(Queue queue, DirectExchange exchange) {
        return BindingBuilder.bind(queue).to(exchange).with("rpc");
    }

    @Bean
    public Jackson2JsonMessageConverter messageConverter() {
        return new Jackson2JsonMessageConverter();
    }

    public static void main(String[] args) {
```

```
        SpringApplication.run(HelloServerApplication.class, args);
    }
}
```

我们在 `HelloServerApplication` 中定义了一个名为 `rpc-queue` 的 `Queue`，也定义了一个名为 `rpc-exchange` 的 `DirectExchange`，此外还创建了 `Queue` 与 `DirectExchange` 之间一个名为 `rpc` 的 `Binding` 关系（或称为 `Routing Key`）。同时，还定义了一个 `Jackson2JsonMessageConverter`，用于实现消息对象与 JSON 字符串的序列化与反序列化操作。

以下是 RPC 服务端需要执行的业务逻辑，我们需要使用 RabbitMQ 提供的 `@RabbitListener` 注解并监听指定队列中的请求消息。

```
package demo.msa.rabbitmq.hello.server;

import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;

@Component
public class HelloServer {

    @RabbitListener(queues = "rpc-queue")
    public String receive(String message) {
        return "hello " + message;
    }
}
```

至此，RPC 服务端开发结束，下面进入 RPC 客户端开发。

RPC 客户端通过 `Exchange` 来发送请求消息，并通过 `Exchange` 路由到的 `Queue` 来获取应答消息。仍然从 RPC 客户端的 `Spring Boot` 应用程序启动类开始。

```
package demo.msa.rabbitmq.hello.client;

import org.springframework.amqp.core.DirectExchange;
import org.springframework.amqp.support.converter.Jackson2JsonMessageConverter;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
```

```
import javax.annotation.PostConstruct;

@SpringBootApplication
public class HelloClientApplication {

    @Autowired
    private HelloClient helloClient;

    @Bean
    public DirectExchange exchange() {
        return new DirectExchange("rpc-exchange");
    }

    @Bean
    public Jackson2JsonMessageConverter messageConverter() {
        return new Jackson2JsonMessageConverter();
    }

    @PostConstruct
    public void init() {
        String result = helloClient.send("world");
        System.out.println(result);
    }

    public static void main(String[] args) {
        SpringApplication.run(HelloClientApplication.class, args).close();
    }
}
```

在 `HelloClientApplication` 中注入了 `HelloClient`（下面会完成该类），定义了一个与 `RPC` 服务端同名的 `DirectExchange`，此外还定义了一个与 `RPC` 服务端一样的 `Jackson2JsonMessageConverter`。

以下是 `RPC` 客户端发送请求消息并接收应答消息的过程，我们注入了 `RabbitTemplate` 与 `DirectExchange`，并通过调用 `RabbitTemplate` 对象的 `convertSendAndReceive()` 方法来执行 `RPC` 调用过程，可见这个方法是同步的，需要等待 `RPC` 服务端的响应结果。

```
package demo.msa.rabbitmq.hello.client;
```

```
import org.springframework.amqp.core.DirectExchange;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class HelloClient {

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @Autowired
    private DirectExchange exchange;

    public String send(String message) {
        return (String) rabbitTemplate.convertSendAndReceive(exchange.getName(),
"rpc", message);
    }
}
```

分别运行 `HelloServerApplication` 与 `HelloClientApplication` 应用程序，可观察 RPC 调用结果。

如果不出意外的话，我们可在 `HelloServerApplication` 应用程序的控制台中看到输出了 `hello world` 字符串，此时表示 RPC 调用成功。

需要补充说明的是，RabbitMQ 提供了以下四种 Exchange。

(1) **Direct Exchange**: 通过 **Routing Key** 将消息从 **Exchange** 传递到指定的 **Queue**，它是 RabbitMQ 中默认的 **Exchange**。

(2) **Topic Exchange**: 可在 **Routing Key** 中使用带有 `*` 或 `#` 的通配符来匹配具体的 **Queue**，常用于指定规则的消息路由。

(3) **Fanout Exchange**: 无须 **Routing Key** 就能将消息传递到与 **Exchange** 对接的所有 **Queue** 中，常用于一对多的消息广播模式。

(4) **Headers Exchange**: 不再使用 **Routing Key** 来实现消息路由，而是通过消息头部进行操作，该类型一般很少用到。

此外，以上代码中用到的 `org.springframework.amqp` 包来自于 Spring AMQP 开源项目，它是 Spring 官方提供的一个子项目，提供了一套对基于 AMQP 协议的代码封装，便于我们更加高效地使用 RabbitMQ。

Spring AMQP: <http://projects.spring.io/spring-amqp/>。

可能大家已经发现了, 在 RPC 服务端与客户端代码中, 存在一些可以被抽取出来的代码片段, 比如 Queue、DirectExchange、Binding、Jackson2JsonMessageConverter 等代码, 此外, 发送消息与接收消息的 API 似乎也能做一些封装, 或者做一些标准。因此, 我们希望对 RabbitMQ 的 RPC 代码框架进行封装, 让 RPC 开发过程变得更加简单。

5.2.3 封装 RabbitMQ 的 RPC 代码框架

为了便于代码使用, 我们分别创建了两个 Maven 项目: rabbitmq-rpc-server 与 rabbitmq-rpc-client, 它们分别封装 RPC 服务端与客户端中需要剥离出来相关代码。

首先我们一起来开发 rabbitmq-rpc-server 项目, 它用于封装 RPC 服务端框架。

我们创建一个名为 RpcServerConfig 的类, 该类中带有 Spring 提供的 @Configuration 注解, 表示该类用于 Spring Bean 的配置, 随后将 HelloServerApplication 类中关于 Queue、DirectExchange、Binding、Jackson2JsonMessageConverter 等代码都抽取出来, 并将其放入 RpcServerConfig 类中。在 Spring Boot 应用程序启动时将会扫描带有 @Configuration 注解的类, 并解析其中的 @Value 与 @Bean 注解, 实现依赖注入与反射调用。

```
package demo.msa.rabbitmq.rpc.server;

import org.springframework.amqp.core.Binding;
import org.springframework.amqp.core.BindingBuilder;
import org.springframework.amqp.core.DirectExchange;
import org.springframework.amqp.core.Queue;
import org.springframework.amqp.support.converter.Jackson2JsonMessageConverter;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class RpcServerConfig {

    @Value("${rabbitmq.queue-name:rpc-queue}")
    private String queueName;

    @Value("${rabbitmq.exchange-name:rpc-exchange}")
    private String exchangeName;
```

```
@Value("${rabbitmq.routing-key:rpc}")
private String routingKey;

@Bean
public Queue queue() {
    return new Queue(queueName);
}

@Bean
public DirectExchange exchange() {
    return new DirectExchange(exchangeName);
}

@Bean
public Binding binding(Queue queue, DirectExchange exchange) {
    return BindingBuilder.bind(queue).to(exchange).with(routingKey);
}

@Bean
public Jackson2JsonMessageConverter messageConverter() {
    return new Jackson2JsonMessageConverter();
}
}
```

除了可以将一部分 RabbitMQ 基础代码进行封装，我们还能 RPC 服务端的处理类 `HelloServer` 做一个接口规范。我们定义了一个名为 `RpcReceiver<I, O>` 的接口，它拥有一个 `receive()` 方法与两个泛型参数，`I` 表示输入参数类型，`O` 表示输出结果类型，通过实现该接口来处理 RPC 服务端中接收到的消息。

```
package demo.msa.rabbitmq.rpc.server;

public interface RpcReceiver<I, O> {

    O receive(I message);
}
```

我们改造以前的 `HelloServer` 类，让它实现 `RpcReceiver<I, O>` 接口，`receive()` 方法除了添加

`@Override` 注解，无须做任何改动。

```
package demo.msa.rabbitmq.hello.server;

import demo.msa.rabbitmq.rpc.server.RpcReceiver;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;

@Component
public class HelloServer implements RpcReceiver<String, String> {

    @Override
    @RabbitListener(queues = "rpc-queue")
    public String receive(String message) {
        return "hello " + message;
    }
}
```

由于 `rabbitmq-rpc-server` 是独立的模块，`rabbitmq-hello-server` 项目仍然需要扫描到该模块，因此需要在 `HelloServerApplication` 类的 `@SpringBootApplication` 注解中添加 `scanBasePackages` 属性，并指定到能够扫描 `rabbitmq-rpc-server` 与 `rabbitmq-hello-server` 的基础包名。

```
package demo.msa.rabbitmq.hello.server;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication(scanBasePackages = "demo.msa.rabbitmq")
public class HelloServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(HelloServerApplication.class, args);
    }
}
```

实际上我们的 `application.proerties` 配置文件无须添加任何配置，因为所有的 `@Value` 注解中都带有配置项的默认值，例如，`@Value("${rabbitmq.queue-name:rpc-queue}")`，它表明从 `application.proerties` 配置文件读取名为 `rabbitmq.queue-name` 的配置项，若该配置项不存在，则该配置项的默认值为 `rpc-queue`。如果大家不喜欢框架中提供的配置项默认值，那么可以在

application.proerties 配置文件重新做出配置。

```
rabbitmq.queue-name=rpc-queue
rabbitmq.exchange-name=rpc-exchange
rabbitmq.routing-key=rpc
```

RPC 服务端框架开发完毕，接下来我们一起来开发 `rabbitmq-rpc-client` 项目，它用于封装 RPC 客户端框架。

在 RPC 客户端只需要知道 `Exchange` 与 `RoutingKey` 即可，无须知道具体的 `Queue`，我们同样创建一个与 RPC 服务端类似的 `RpcClientConfig` 配置类，将以前放在 `HelloClientApplication` 类中关于 `DirectExchange` 与 `Jackson2JsonMessageConverter` 的配置都搬运过来。

```
package demo.msa.rabbitmq.rpc.client;

import org.springframework.amqp.core.DirectExchange;
import org.springframework.amqp.support.converter.Jackson2JsonMessageConverter;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class RpcClientConfig {

    @Value("${rabbitmq.exchange-name:rpc-exchange}")
    private String exchangeName;

    @Bean
    public DirectExchange exchange() {
        return new DirectExchange(exchangeName);
    }

    @Bean
    public Jackson2JsonMessageConverter messageConverter() {
        return new Jackson2JsonMessageConverter();
    }
}
```

除了搬运配置，我们还能将 RPC 客户端的调用过程进行封装，也就是说，将以前 `HelloClient`

类中关于 `RabbitTemplate` 与 `DirectExchange` 的相关代码移动到以下 `RpcSender<I, O>` 类中。

```
package demo.msa.rabbitmq.rpc.client;

import org.springframework.amqp.core.DirectExchange;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class RpcSender<I, O> {

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @Autowired
    private DirectExchange exchange;

    @Value("${rabbitmq.routing-key:rpc}")
    private String routingKey;

    @SuppressWarnings("unchecked")
    public O send(I message) {
        return (O) rabbitTemplate.convertSendAndReceive(exchange.getName(), routingKey,
message);
    }
}
```

`RpcSender<I, O>` 类与之前的 `RpcReceiver<I, O>` 接口类似，都带有两个泛型参数，而且这两个泛型参数的意义完全相同，`I` 表示输入参数类型，`O` 表示输出结果类型，此外还在 `send()` 方法中对 `RabbitTemplate` 的调用过程进行了封装，这样 `HelloClient` 只需注入 `RpcSender`，并指定两个具体的泛型参数，就能调用它的 `send()` 方法来实现 RPC 调用。

```
package demo.msa.rabbitmq.hello.client;

import demo.msa.rabbitmq.rpc.client.RpcSender;
import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.stereotype.Component;

@Component
public class HelloClient {

    @Autowired
    private RpcSender<String, String> rpcSender;

    public String send(String message) {
        return rpcSender.send(message);
    }
}
```

此时的 `HelloClientApplication` 代码就显得比较干净了，所有关于 RabbitMQ 的一切都放入了 RPC 客户端框架中。

```
package demo.msa.rabbitmq.hello.client;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

import javax.annotation.PostConstruct;

@SpringBootApplication(scanBasePackages = "demo.msa.rabbitmq")
public class HelloClientApplication {

    @Autowired
    private HelloClient helloClient;

    @PostConstruct
    public void init() {
        String result = helloClient.send("world");
        System.out.println(result);
    }

    public static void main(String[] args) {
        SpringApplication.run(HelloClientApplication.class, args).close();
    }
}
```

```

    }
}

```

当然,也能在RPC客户端的`application.properties`配置文件中对默认的`Exchange`与`RoutingKey`配置项进行改写。

```

rabbitmq.exchange-name=rpc-exchange
rabbitmq.routing-key=rpc

```

至此,基于RabbitMQ的RPC框架已搭建完毕,大家可以再次运行装修过的RPC服务端与客户端,并观察程序运行的结果是否与以前一致。

以上只是一个简单的封装,但能为我们的应用开发屏蔽掉许多底层的细节,让框架代码与业务代码相分离,这正是框架为业务带来的价值。

5.3 解决分布式事务问题

在微服务架构下,每个服务可以分布在不同的机器上独立运行,对外提供调用接口,服务之间还能相互调用。这样的架构虽然非常灵活,但是当服务之间发生调用时,就会产生分布式事务问题,如何解决该问题则是微服务架构中的难题。在Java领域中,我们首先能想到的是JTA (Java Transaction API, Java 事务 API),它是JavaEE所提出的分布式事务规范,实际上就是一组标准的API接口,但没有具体的实现。Spring与JOTM、Atomikos等框架相集成虽然可以实现JTA分布式事务,但使用起来过于复杂,性能也会有所降低,而且仅用于Java应用系统,这些问题违背了我们所提倡的轻量级微服务架构的初衷,JTA既不轻量级,也不跨平台。其实解决分布式事务的本质是解决数据一致性,我们完全没必要使用繁重的技术框架去保证数据的“强一致性”,这样不仅会导致技术实现过于复杂,而且还会降低整个系统的调用开销,我们更愿意尝试做到的是数据的“最终一致性”。

Event-Sourcing (事件溯源)是一种实现数据最终一致性问题的有效解决方案,我们可将它与MQ相结合,轻松解决分布式事务问题,那么Event-Sourcing究竟是什么呢?

5.3.1 什么是 Event-Sourcing

在微服务这个概念诞生之前,其实Event-Sourcing就已经出现了,它是一种基于“事件溯源”的解决方案,一般将它应用在领域对象模型中。在一个对象从创建到销毁的整个生命周期中,会产生大量的事件(Event),然而每种事件都有自己所属的事件类型(Event Type)。事件类型是可以枚举的,而事件无法枚举,事件包含时间、事件类型、模型等信息。例如,“创建”

是一种事件类型，而“在某时刻创建一个产品”则是一个事件。

一般情况下，我们只是将对象的最终状态记录到数据库中，而不会去记录每个对象的历史事件变更。Event-Sourcing 要求我们在记录对象状态的同时，还要去记录对象所发生的一系列事件，而且这些事件需要随时间的先后顺序依次记录到数据库中。也就是说，数据库中不仅包含对象所属的模型表，还包含对象所产生事件的事件表。当创建、修改、删除一条模型对象时（查询一般不考虑），不仅需要将此对象插入模型表，还要记录一条对应的事件到事件表中，这个过程称为“事件记录”。

例如，我们在 Foo Service 中创建了一个 Foo 对象，此时需将 Foo 对象插入到 Foo Table 中，同时记录一条“CREATE Foo”的事件到 Event Table 中，如图 5-15 所示。

每个模型对象都拥有一个唯一的 ID，称为 Model ID。对于事件对象而言，也同样拥有一个唯一的 ID，称为 Event ID。我们可通过 Event ID 在 Event Table 中查询对应的 Model ID，这一过程称为“事件溯源”。

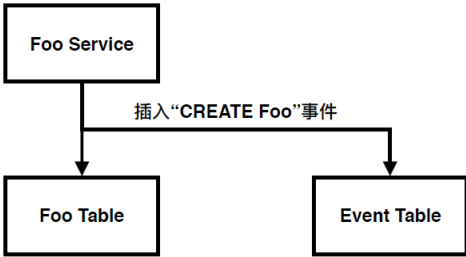


图 5-15 事件记录过程

例如，我们获得了一个 Event ID，就能在 Event Table 中根据 Event ID 查询 Foo ID，随后可在 Foo Table 中通过 Foo ID 操作对应的 Foo 对象，如图 5-16 所示。

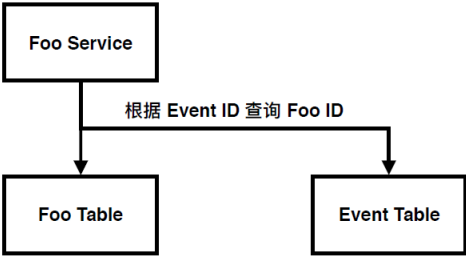


图 5-16 事件溯源过程

也就是说，在 Event-Sourcing 中，我们首先要进行的是事件记录，随后才能做到事件溯源。在事件记录过程中，我们先操作模型表，再操作事件表；而在事件溯源过程中，我们先操作事件表，再操作模型表，这两个过程所操作数据表的步骤正好相反。

对于事件表而言，它的结构比较固定，但也能根据实际情况灵活设计，但大体包括以下基础字段。

- ID: 表示 Event ID，具备唯一性。
- Event Type: 表示事件类型，例如，CREATE、UPDATE、DELETE 等。
- Model Name: 表示模型名称，例如，Foo、Bar 等。
- Model ID: 表示对应的模型对象 ID，具备唯一性。
- Created Time: 表示创建事件的具体时间，一般精确到毫秒级。

可见，Event-Sourcing 的核心原理并不难理解，那么到底 Event-Sourcing 与分布式事务有何关系呢？又该如何利用 Event-Sourcing 来解决分布式事务问题呢？

实际上，单纯依靠 Event-Sourcing 的设计理念是无法解决分布式事务问题的，我们需要将 Event-Sourcing 与 MQ 结合起来才能发挥价值，从而实现分布式事务控制。

5.3.2 使用 Event-Sourcing 与 MQ 实现分布式事务控制

假如我们有 Foo Service 与 Bar Service 两个服务，它们分别运行在不同的进程中，或不同的机器上，总之它们是分布式在整个微服务架构中的两个普通的服务。首先 Foo Service 插入一条 Foo 对象到 Foo Table 中，随后通过 MQ 的方式去调用 Bar Service，最后 Bar Service 也插入了一条 Bar 对象到 Bar Table 中，这是一个经典的分布式调用场景，如图 5-17 所示。

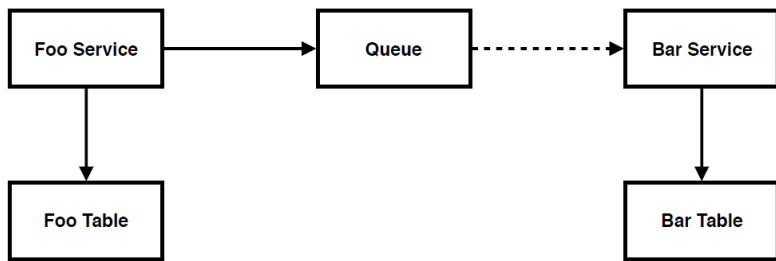


图 5-17 分布式调用场景

若 Bar Service 在插入 Bar 对象时出现了异常，则首先需要回滚自己的事务，同时 Foo Service 也要回滚曾经所提交的事务，这两个是事务是分布式的，还要确保是原子性的。我们接下来将基于该场景实现分布式事务控制。

在解决分布式事务控制之前，有必要对事件相关对象与操作进行一些代码封装，我们首先定义下面几个类。

- Event: 用于封装事件相关字段。

- **EventType**: 用于定义各种事件类型。
- **EventManager**: 用于封装事件相关操作。

下面是 **Event** 类的相关代码，它是一个普通的 **POJO** 类。

```
package demo.msa.event;

import java.util.UUID;

public class Event {

    private String id;
    private EventType eventType;
    private String modelName;
    private String modelId;
    private long createTime;

    public Event() {
    }

    public Event(EventType eventType, String modelName, String modelId) {
        this.id = UUID.randomUUID().toString();
        this.eventType = eventType;
        this.modelName = modelName;
        this.modelId = modelId;
        this.createTime = System.currentTimeMillis();
    }

    // 省略 getter/setter 方法
}
```

以上 **Event** 类与 **Event Table** 相映射，为了简化整个过程，由于 **id** 需要保证唯一性，我们简单地使用 **UUID** 来随机生成，**createTime** 由当前系统时间决定。

在 **Event** 类中包括一个 **EventType** 属性，它对应一个简单的枚举类。

```
package demo.msa.event;

public enum EventType {
```

```
CREATE, UPDATE, DELETE
}
```

目前我们打算完成 `EventManager` 类中的相关操作方法，只为该类搭建一个代码框架。

```
package demo.msa.event;

import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.jdbc.core.JdbcTemplate;

public class EventManager {

    private final JdbcTemplate jdbcTemplate;
    private final RabbitTemplate rabbitTemplate;

    public EventManager(JdbcTemplate jdbcTemplate, RabbitTemplate rabbitTemplate) {
        this.jdbcTemplate = jdbcTemplate;
        this.rabbitTemplate = rabbitTemplate;
    }

    // TODO: 此处将提供相关操作方法
}
```

`EventManager` 类需要传入 `JdbcTemplate` 对象，用于向 `Event Table` 中插入数据，还需传入 `RabbitTemplate` 对象，用于向 `RabbitMQ` 消息队列中写入消息。

在我们的案例中，使用了以下三张数据表，它们的表结构如下所示。

```
CREATE TABLE `event` (
  `id` char(36) NOT NULL DEFAULT '',
  `event_type` varchar(100) DEFAULT NULL,
  `model_name` varchar(100) DEFAULT NULL,
  `model_id` char(36) DEFAULT NULL,
  `created_time` bigint(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `foo` (
```

```
`id` char(36) NOT NULL DEFAULT '',
`name` varchar(100) DEFAULT NULL,
PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

```
CREATE TABLE `bar` (
  `id` char(36) NOT NULL DEFAULT '',
  `name` varchar(100) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

应用程序使用 Maven 来构建，以下是 pom.xml 配置文件中所需的 Maven 依赖。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
```

...

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.5.2.RELEASE</version>
</parent>
```

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
```

```

        <artifactId>spring-boot-starter-jdbc</artifactId>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
    </dependency>
</dependencies>

</project>

```

现在我们将通过图文与代码的方式演示整个实现过程，该过程只需以下三步。

第一步：操作模型表与事件表，并将事件写入消息队列。

在 Foo Service 中插入一条 Foo 对象到 Foo Table 中，同时创建一个名为“CREATE Foo”的事件到 Event Table 中，这两个 JDBC 操作很容易确保在同一个事务中提交。此外，在事务提交之前，可将插入 Event Table 中的 Event 对象插入 Success Queue（成功队列）中，如图 5-18 所示。

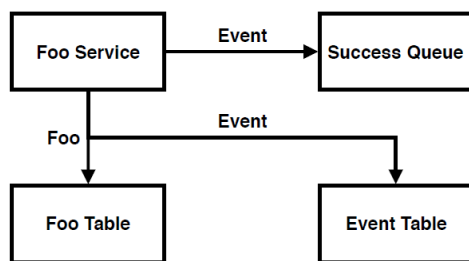


图 5-18 在 Foo Service 中记录事件

可见，以上过程实际上包括以下四个任务。

- (1) 将 Foo 对象插入模型表中。
- (2) 创建一个 Event 对象。
- (3) 将 Event 对象插入事件表中。
- (4) 将 Event 对象写入成功队列中。

下面，我们将以上四个任务封装在 FooService 类的 insertFoo()方法中。

```

package demo.msa.foo.service;

import demo.msa.event.Event;

```

```
import demo.msa.event.EventManager;
import demo.msa.event.EventType;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.UUID;

@Service
public class FooService {

    private JdbcTemplate jdbcTemplate;
    private EventManager eventManager;

    @Autowired
    public FooService(JdbcTemplate jdbcTemplate, EventManager eventManager) {
        this.jdbcTemplate = jdbcTemplate;
        this.eventManager = eventManager;
    }

    @Transactional
    public void insertFoo(String name) {
        String fooId = UUID.randomUUID().toString();
        try {
            // 将 Foo 对象插入模型表中
            jdbcTemplate.update(
                "INSERT INTO foo (id, name) VALUES (?, ?)",
                fooId, name
            );
        } finally {
            // 创建一个 Event 对象
            Event event = new Event(EventType.CREATE, "Foo", fooId);
            // 将 Event 对象插入事件表中
            eventManager.insertEvent(event);
            // 将 Event 对象写入成功队列中
            eventManager.sendEventQueue("foo-success-queue", event);
        }
    }
}
```

```

    }
}

```

由于在 `insertFoo()` 方法上带有 `@Transactional` 注解, 那么 Spring 框架将确保该方法带有事务性, 也就是说, 方法内部包含的所有数据库操作将在同一个事务中提交。在 `try` 代码块中我们完成了插入 Foo 对象的数据库操作, 在 `finally` 代码块中完成了插入 Event 对象的事件记录操作, 最后将该 Event 对象写入一个名为 “foo-success-queue” 的消息队列中。

第二步: 从消息队列中获取事件, 并操作模型表, 若有异常情况, 则将源事件再次写入消息队列。

对于 Success Queue 而言, Bar Service 是一个消费者, 它将从 Success Queue 中获取 Event 对象, 并完成自己的业务逻辑, 即插入一个 Bar 对象到 Bar Table 中。但如果在完成业务操作的过程中, 发生了任何异常行为, 此时我们将从 Success Queue 中接收到的 Event 对象写入 Failure Queue (失败队列) 中, 如图 5-19 所示。

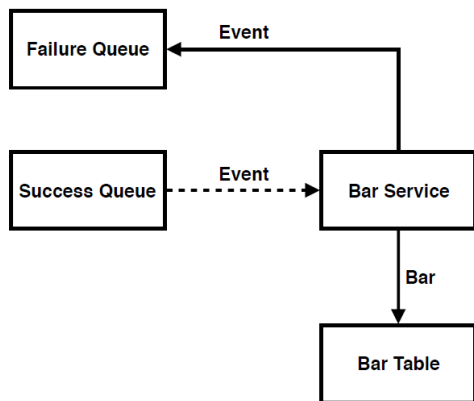


图 5-19 在 Bar Service 抛出异常

可见, 以上过程实际上包括以下四个任务。

- (1) 从成功队列中获取 Event 对象。
- (2) 将 Bar 对象插入模型表中。
- (3) 故意抛出异常。
- (4) 将 Event 对象写入失败队列中。

下面, 我们将以上四个任务封装在 BarService 类的 `handleFooSuccess()` 方法中。

```
package demo.msa.bar.service;
```

```
import demo.msa.event.Event;
import demo.msa.event.EventManager;
import org.springframework.amqp.AmqpRejectAndDontRequeueException;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.UUID;

@Service
public class BarService {

    private JdbcTemplate jdbcTemplate;
    private EventManager eventManager;

    @Autowired
    public BarService(JdbcTemplate jdbcTemplate, EventManager eventManager) {
        this.jdbcTemplate = jdbcTemplate;
        this.eventManager = eventManager;
    }

    @Transactional
    @RabbitListener(queues = "foo-success-queue")
    public void handleFooSuccess(Event event) { // 从成功队列中获取 Event 对象
        try {
            String barId = UUID.randomUUID().toString();
            // 将 Bar 对象插入模型表中
            jdbcTemplate.update(
                "INSERT INTO bar (id, name) VALUES (?, ?)",
                barId, "bar"
            );
            // 故意抛出异常
            throw new RuntimeException();
        } catch (Exception e) {
            // 将 Event 对象写入失败队列中
            eventManager.sendEventQueue("bar-failure-queue", event);
        }
    }
}
```



```

// 让事务进行回滚
throw new AmqpRejectAndDontRequeueException(e);
}
}
}

```

在 `handleFooSuccess()` 方法上同样也带有 `@Transactional` 注解，此外还带有 `@RabbitListener` 注解，它表示从指定的消息队列中获取消息，此时就需要从 `foo-success-queue` 队列中获取 `Event` 对象。在 `try` 代码块中我们完成了业务操作，随后立即抛出了一个 `RuntimeException` 异常，此处仅用于制造方法内部的异常行为，当前线程将进入 `catch` 代码块。在 `catch` 代码块中我们首先将 `Event` 对象写入一个名为“`bar-failure-queue`”的消息队列中，随后对外抛出一个 Spring AMQP 框架提供的 `AmqpRejectAndDontRequeueException` 异常，并将当前捕获的 `RuntimeException` 异常传入其中。

需要注意的是，如果不在 `catch` 中抛出 `AmqpRejectAndDontRequeueException` 异常，虽然 JDBC 事务可以回滚，但 `Event` 对象将重新进入 `foo-success-queue` 队列，此时 Spring AMQP 框架将再次调用 `handleFooSuccess()` 方法，又将重复执行同样的代码，导致此过程不断循环，直到超出 RabbitMQ 队列所限制的 `Requeue` 次数，这明显不是我们所希望的行为。当我们抛出了 `AmqpRejectAndDontRequeueException` 异常时，`Event` 对象将不再进入 `foo-success-queue` 队列，而是进入 DLQ（Dead Letter Queue，死信队列）中，进入其中的消息将被丢弃，永远不会被消费。

第三步：从消息队列中获取事件，操作事件表与模型表（即“事件溯源”过程）。

此时，`Foo Service` 不再是消息的生产者，而是消息的消费者，它需从 `Failure Queue` 从取回曾经发送到 `Success Queue` 中的 `Event` 对象，并根据 `Event ID` 到 `Event Table` 中查询出 `Foo ID`，随后根据 `Foo ID` 到 `Foo Table` 中删除曾经创建的 `Foo` 对象，如图 5-20 所示。

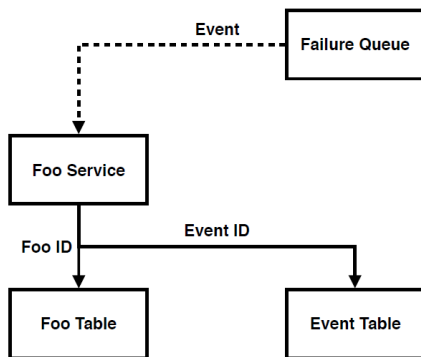


图 5-20 在 `Foo Service` 中事件溯源

可见，以上过程实际上包括以下三个任务。

- (1) 从失败队列中获取 Event 对象。
- (2) 根据 Event ID 从事件表中获取 Foo ID。
- (3) 根据 Foo ID 从模型表中删除对应的记录。

下面，我们将以上三个任务封装在 FooService 类的 handleBarFailure()方法中。

```
package demo.msa.foo.service;

...
import org.springframework.amqp.rabbit.annotation.RabbitListener;
...

import java.util.UUID;

@Service
public class FooService {

    ...

    @Transactional
    @RabbitListener(queues = "bar-failure-queue")
    public void handleBarFailure(Event event) { // 从失败队列中获取 Event 对象
        // 根据 Event ID 从事件表中获取 Foo ID
        String fooId = eventManager.queryModelId(event.getId());
        // 根据 Foo ID 从模型表中删除对应的记录
        jdbcTemplate.update(
            "DELETE FROM foo WHERE id = ?",
            fooId
        );
    }
}
```

此处的 handleBarFailure()方法与上面的 handleFooSuccess()方法类似，同样具有 @Transactional 与 @RabbitListener 注解，与此不同的是，现在无须处理任何异常，而是根据 Event ID 去查询 Foo ID，并根据 FooID 去删除 Foo 对象。可见，此时只是为了进行一个与之前正常业务逻辑相对应的逆向操作，目的是为了确保数据库中没有任何无效数据，从而做到了数据的最终一致性。

需要注意的是，我们的案例中仅包含 INSERT 语句，它的逆向操作是 DELETE 语句，对于 UPDATE 语句的逆向操作也是 UPDATE 语句，但对于 DELETE 语句而言，我们更倾向于逻辑删除（更新删除标记字段），而不是物理删除，因此 DELETE 语句的逆向操作也是 UPDATE 语句。

以上 FooService 与 BarService 中调用了 EventManager 类所提供的 insertEvent()、queryModelId()、sendEventQueue()等方法，最后我们将这些方法补充到 EventManager 类中。

```
package demo.msa.event;

...

public class EventManager {

    ...

    public void insertEvent(Event event) {
        jdbcTemplate.update(
            "INSERT INTO event (id, event_type, model_name, model_id,
created_time) VALUES (?, ?, ?, ?, ?)",
            event.getId(), event.getEventType().toString(), event.getModelName(),
event.getModelId(), event.getCreatedTime()
        );
    }

    public String queryModelId(String eventId) {
        return jdbcTemplate.queryForObject(
            "SELECT model_id FROM event WHERE id = ?",
            String.class,
            eventId
        );
    }

    public void sendEventQueue(String queueName, Event event) {
        rabbitTemplate.convertAndSend(queueName, event);
    }
}
```

接下来，我们分别为 Foo Service 与 Bar Service 创建对应 Spring Boot 应用程序启动类。

以下是 Foo Service 的 Spring Boot 应用程序启动类 `FooApplication`，其中包括了两个消息队列以及所需相关 Spring Bean 的实例化代码。

```
package demo.msa.foo;

import demo.msa.event.EventManager;
import org.springframework.amqp.core.Queue;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.amqp.support.converter.Jackson2JsonMessageConverter;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.jdbc.core.JdbcTemplate;

@SpringBootApplication
public class FooApplication {

    @Bean
    public Queue successQueue() {
        return new Queue("foo-success-queue");
    }

    @Bean
    public Queue failureQueue() {
        return new Queue("bar-failure-queue");
    }

    @Bean
    public Jackson2JsonMessageConverter messageConverter() {
        return new Jackson2JsonMessageConverter();
    }

    @Bean
    public EventManager eventManager(JdbcTemplate jdbcTemplate, RabbitTemplate rabbitTemplate) {
        return new EventManager(jdbcTemplate, rabbitTemplate);
    }
}
```

```

    public static void main(String[] args) {
        SpringApplication.run(FooApplication.class, args);
    }
}

```

Bar Service 所对应的 BarApplication 代码与以上 FooApplication 类似，请大家自行完成。以下是 Foo Service 应用程序的 application.properties 配置文件，Bar Service 与此类似。

```

server.port=8080

spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/demo?useSSL=false
spring.datasource.username=root
spring.datasource.password=root

spring.rabbitmq.addresses=localhost:5672
spring.rabbitmq.username=admin
spring.rabbitmq.password=admin

```

为了便于查看 JdbcTemplate 类所执行的 SQL 语句，我们可在 application.properties 配置文件中提供如下配置，将 JdbcTemplate 的日志级别调整为 debug。

```
logging.level.org.springframework.jdbc.core.JdbcTemplate=debug
```

最后，我们在 Foo Service 应用中通过一个 Controller 来调用调用 FooService 类的 insertFoo() 方法。

```

package demo.msa.foo.controller;

import demo.msa.foo.service.FooService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import javax.websocket.server.PathParam;

@RestController
public class FooController {

```

```

private FooService fooService;

@Autowired
public FooController(FooService fooService) {
    this.fooService = fooService;
}

@GetMapping("/foo")
public void foo(@PathParam("name") String name) {
    fooService.insertFoo(name);
}
}

```

现在我们分别启动 `FooApplication` 与 `BarApplication`，在浏览器中访问 `http://localhost:8080/foo?name=a`，随后可立即观察这两个应用程序的控制台中的日志输出。

我们可在 `BarApplication` 的控制台中可看到如下信息，表示插入一条 `Bar` 记录后，就抛出了 `RuntimeException` 异常，该异常通过 `AmqpRejectAndDontRequeueException` 异常继续向外抛出。

```

Executing prepared SQL update
Executing prepared SQL statement [INSERT INTO bar (id, name) VALUES (?, ?)]
SQL update affected 1 rows
Execution of Rabbit message listener failed.
org.springframework.amqp.rabbit.listener.exception.ListenerExecutionFailedException: Listener method 'public void demo.msa.bar.service.BarService.handleFooSuccess(demo.msa.event.Event)' threw exception
...
Caused by: org.springframework.amqp.AmqpRejectAndDontRequeueException:
java.lang.RuntimeException
...
Caused by: java.lang.RuntimeException: null
...

```

如果此时 `MySQL` 数据库中 `bar` 表未插入任何数据，则说明 `Bar Service` 的事务回滚了。

同时，我们可观察到 `FooApplication` 的控制台，首先向 `foo` 表中插入了一条记录，随后向 `event` 表插入了一条记录，最后通过 `id` 从 `event` 表中查询到 `model_id`（即 `Foo ID`），并在 `foo` 表中根据 `Foo ID` 在 `foo` 表中删除对应的记录。

```

Executing prepared SQL update

```

```

Executing prepared SQL statement [INSERT INTO foo (id, name) VALUES (?, ?)]
SQL update affected 1 rows
Executing prepared SQL update
Executing prepared SQL statement [INSERT INTO event (id, event_type,
model_name, model_id, created) VALUES (?, ?, ?, ?, ?)]
SQL update affected 1 rows
Executing prepared SQL query
Executing prepared SQL statement [SELECT model_id FROM event WHERE id = ?]
Executing prepared SQL update
Executing prepared SQL statement [DELETE FROM foo WHERE id = ?]
SQL update affected 1 rows

```

此时可观察 MySQL 数据库中的三张表，只有 event 表中有数据，foo 与 bar 表中均没有任何数据。

至此，我们所面临的分布式事务问题已得到解决。当然这只是一个简单的案例，仍然还有很多需要优化的空间，此仅作为参考。

整个分布式事务控制所涉及的三个步骤可通过一副流程图来描绘，如图 5-21 所示。

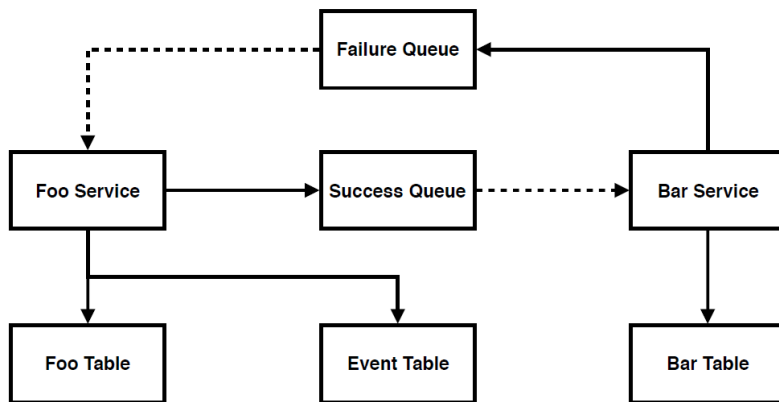


图 5-21 分布式事务控制流程图

分布式事务虽然复杂，但我们无须追求数据的强一致性，利用 Event-Sourcing 与 MQ 就能确保数据的最终一致性，从而控制了分布式事务。但我们想要强调的是，我们应做到尽量避免分布式事务，通过合理的切分微服务边界可解决大部分分布式事务问题，对于少量情况而言，我们才使用以上方案来解决。总之，处理分布式事务有一定的挑战，有挑战就有风险，我们能避免分布式事务就尽量避免吧。

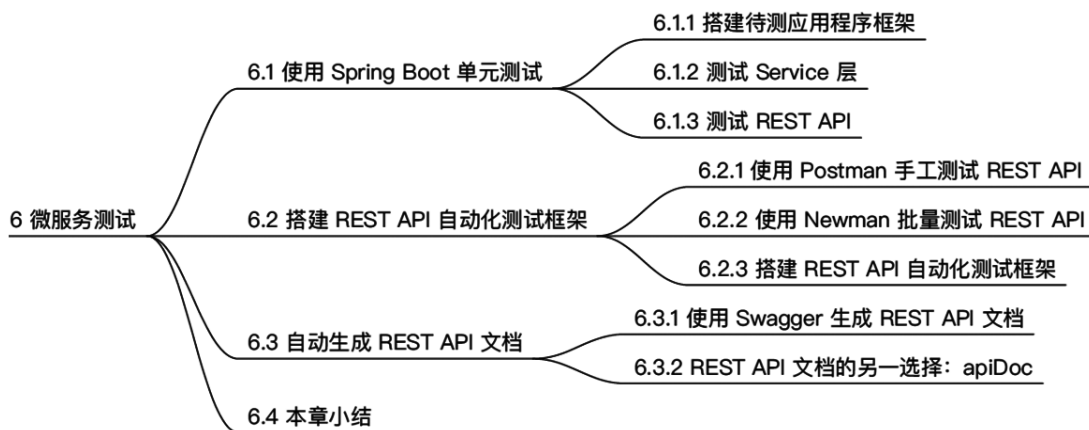
5.4 本章小结

本章使用了消息队列的异步方式了解耦微服务调用问题，这也是业界解决服务之间耦合问题的首选方案。首先我们对比了 ActiveMQ 与 RabbitMQ 这两款经典的开源消息队列，为了追求更高的性能，我们选择了 RabbitMQ 作为我们轻量级微服务架构的“消息中心”。随后我们使用了 RabbitMQ 来实现“请求应答模式”，并通过 RabbitMQ 实现了 RPC 同步调用，可见 RabbitMQ 不仅有优秀的性能，还有强大的功能。最后我们使用 Event-Sourcing 架构模式并将其与 RabbitMQ 相结合，巧妙地解决分布式事务问题，但我们仍然建议大家尽可能通过合理划分服务边界来化解服务之间的耦合现象，从而避免分布式事务问题。

下一章我们将转向微服务测试方面，目标是让微服务对外提供的 REST API 具备更高的稳定性。

6chapter

第 6 章 微服务测试



6.1 使用 Spring Boot 单元测试

我们已使用 Spring Boot 框架将服务开发完毕，在正式投入生产环境之前，一般都会做大量的测试工作，比如单元测试、集成测试、压力测试、安全测试等。对于开发人员而言，单元测试是最为重要的测试环节之一，它能确保代码的正确性。此外，当代码发生调整后，还能快速进行回归测试，从而确保代码的稳定性。因此，我们有必要先对单元测试进行一些探讨，让单元测试变得更有价值。更值得推荐的做法是，先把应用程序框架搭建起来，不马上开始实现业务逻辑，而是先编写单元测试，通过单元测试来完成业务逻辑。我们鼓励这种“测试先行”的方式，在敏捷开发中，称这种开发模式为“测试驱动开发”，即 TDD (Test-Driven Development)。

本节我们将使用 Spring Boot 单元测试框架，并玩转一些高效的测试工具，让它们为我们所用。

6.1.1 搭建待测应用程序框架

Spring Boot 已经集成了 Spring 的单元测试框架，让单元测试变得更加简单且直接，只需在 Spring Boot 应用程序中添加一个 spring-boot-starter-test 插件的 Maven 依赖即可。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
```

该插件依赖了一些测试框架和类库，比如 spring-test、junit、hamcrest、assertj、jsonpath、jsonassert、mockito 等。

下面我们就来开发一个简单的 Spring Boot 应用程序，并将其作为待测应用程序，以下是 pom.xml 文件中的所有代码。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
```

```
<groupId>demo.msa</groupId>
<artifactId>product</artifactId>
<version>1.0.0</version>

<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.5.2.RELEASE</version>
</parent>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

</project>
```

该应用程序包括一个 **Product** 模型类，它是一个 POJO，带有一些关键性属性。

```
package demo.msa.product.model;

public class Product {

    private long id;
    private String name;
    private int price;
    private long created;

    public Product() {
    }
}
```

```
public Product(long id, String name, int price, long created) {
    this.id = id;
    this.name = name;
    this.price = price;
    this.created = created;
}

// getter/setter ...
}
```

此外该应用程序还拥有另一个 **ProductService** 服务类，其中包括一些业务方法。

```
package demo.msa.product.service;

import demo.msa.product.model.Product;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.Map;

@Service
public class ProductService {

    public Product getProductById(long id) {
        return null;
    }

    public Product createProduct(String name, int price) {
        return null;
    }

    public Product updateProduct(long id, Map<String, Object> fieldMap) {
        return null;
    }

    public boolean deleteProductById(long id) {
        return false;
    }
}
```

```
public List<Product> getProductList() {  
    return null;  
}  
}
```

此时我们并没有对 `ProductService` 类中的业务方法进行具体实现，它看起来更像一个“空架子”，因为我们后续将通过 TDD 开发方式去逐步实现其中的业务逻辑。

如果说 `Service` 层是为了对业务逻辑进行封装，那么 `Controller` 层就是为了调用 `Service` 层并将结果返回给前端。因此，`Controller` 层一定是很“薄”的，不应该带有任何的业务逻辑。

一般情况下，前端通过 REST API 的方式来调用后端的 `Controller` 层，并将返回的 JSON 数据在前端页面中进行渲染，这是前后端分离的基本原则。因此，我们在 `Controller` 层中对前端暴露以下几种有代表性的 REST API。

- 根据 ID 获取产品：GET:/product/{id};
- 创建产品：POST:/product;
- 根据 ID 更新产品：PUT:product/{id};
- 根据 ID 删除产品：DELETE:product/{id};
- 获取所有产品：GET:/product。

以下是 `ProductController` 类的所有代码，我们使用 Spring 框架所提供的 `@RestController`、`@RequestMapping` 等注解，通过它们来完成 REST API 的定义。

```
package demo.msa.product.controller;  
  
import demo.msa.product.model.Product;  
import demo.msa.product.request.ProductRequest;  
import demo.msa.product.response.ProductResponse;  
import demo.msa.product.service.ProductService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.*;  
  
import java.util.List;  
import java.util.Map;  
  
@RestController  
public class ProductController {
```

```
@Autowired
private ProductService productService;

@GetMapping("/product/{id}")
public Product getProductById(@PathVariable("id") long id) {
    return productService.getProductById(id);
}

@PostMapping("product")
public Product createProduct(@RequestBody ProductRequest productRequest) {
    String name = productRequest.getName();
    int price = productRequest.getPrice();
    return productService.createProduct(name, price);
}

@PutMapping("product/{id}")
public Product updateProduct(@PathVariable("id") long id, @RequestBody
Map<String, Object> fieldMap) {
    return productService.updateProduct(id, fieldMap);
}

@DeleteMapping("/product/{id}")
public Product deleteProductById(@PathVariable("id") long id) {
    Product product = productService.getProductById(id);
    return productService.deleteProductById(id) ? product : null;
}

@GetMapping("/product")
public ProductResponse getAllProducts() {
    List<Product> productList = productService.getProductList();
    ProductResponse response = new ProductResponse();
    response.setProductList(productList);
    response.setTotal(productList.size());
    return response;
}
}
```

在 `ProductController` 类中，我们用到了两个数据载体：`ProductRequest` 与 `ProductResponse`。前者用于封装调用 `POST:/product` 请求的请求参数，后者用于封装调用 `GET:/product` 请求的响应结果。

以下是 `ProductRequest` 类的相关代码，它是一个 POJO。

```
package demo.msa.product.request;

public class ProductRequest {

    private String name;
    private int price;

    // 省略 getter/setter 方法
}
```

以下是 `ProductResponse` 类的相关代码，它同样也是一个 POJO。

```
package demo.msa.product.response;

import com.fasterxml.jackson.annotation.JsonProperty;
import demo.msa.product.model.Product;

import java.util.List;

public class ProductResponse {

    @JsonProperty("items")
    private List<Product> productList;

    private int total;

    // 省回 getter/setter 方法
}
```

此时用到了 Jackson 框架提供的 `@JsonProperty` 注解，它用于修改 JSON 序列化的属性映射规则，将 POJO 中的 `productList` 属性映射为 JSON 中的 `items` 属性。

最后只需创建一个 `ProductApplication` 类，就能完成待测应用程序框架。

```
package demo.msa.product;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class ProductApplication {

    public static void main(String[] args) {
        SpringApplication.run(ProductApplication.class, args);
    }
}
```

以上应用程序中，除了 **ProductService** 类没有完成，其他部分的内容基本完成了，接下来我们将使用 TDD 敏捷方法，通过测试来驱动开发。

6.1.2 测试 Service 层

我们首先在 `src/test/java` 目录中创建一个名为 **ProductServiceTest** 的单元测试类，它使用 JUnit 与 Spring Boot 框架所提供的相关注解。

```
package demo.msa.product.test;

import demo.msa.product.service.ProductService;
import org.junit.FixMethodOrder;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.MethodSorters;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@SpringBootTest
@FixMethodOrder(MethodSorters.NAME_ASCENDING)
public class ProductServiceTest {

    @Autowired
```



```
private ProductService productService;

@Test
public void test1_getProductById() throws Exception {
}

@Test
public void test2_createProduct() throws Exception {
}

@Test
public void test3_updateProduct() throws Exception {
}

@Test
public void test4_deleteProductById() throws Exception {
}

@Test
public void test5_getProductList() throws Exception {
}
}
```

我们在类名上使用了`@RunWith`与`@SpringBootTest`注解确保该JUnit单元测试类能与Spring Boot框架进行整合，也就是说，运行该类可识别Spring框架提供的`@Autowired`注解，实现依赖注入特性，也能在JUnit框架中使用Spring框架的其他特性。此外，由于批量运行JUnit测试时，会以随机的方式调用单元测试类中的方法，为了确保方法JUnit框架能按照自然顺序来调用待测方法，我们做了两件事情。首先将所有待测方法的方法名前面增加了一个以编号开头的前缀，其次在类名上使用了JUnit框架所提供`@FixMethodOrder`注解，并将其属性指定为`MethodSorters.NAME_ASCENDING`，此时才能确保所有待测方法可根据字母升序的方式被JUnit框架调用。

在每个方法中来完成我们的单元测试，此时需确保每个方法只做一件测试任务，这样才能将测试做到单元化，这才是单元测试的本质。

```
package demo.msa.product.test;

import demo.msa.product.model.Product;
```

```
import demo.msa.product.service.ProductService;
import org.junit.Assert;
import org.junit.FixMethodOrder;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.MethodSorters;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

@RunWith(SpringRunner.class)
@SpringBootTest
@FixMethodOrder(MethodSorters.NAME_ASCENDING)
public class ProductServiceTest {

    @Autowired
    private ProductService productService;

    @Test
    public void test1_getProductById() throws Exception {
        Product product = productService.getProductById(1);
        Assert.assertNotNull("product is null", product);
    }

    @Test
    public void test2_createProduct() throws Exception {
        Product product = productService.createProduct("iMac", 8000);
        Assert.assertNotNull("product is null", product);
    }

    @Test
    public void test3_updateProduct() throws Exception {
        Map<String, Object> fieldMap = new HashMap<>();
        fieldMap.put("price", 9000);
```

```
Product product = productService.updateProduct(4, fieldMap);
Assert.assertNotNull("product is null", product);
Assert.assertEquals("product.price is wrong", 9000, product.getPrice());
}

@Test
public void test4_deleteProductById() throws Exception {
    boolean result = productService.deleteProductById(4);
    Assert.assertTrue("delete failure", result);
}

@Test
public void test5_getProductList() throws Exception {
    List<Product> productList = productService.getProductList();
    Assert.assertNotNull("productList is null", productList);
    Assert.assertEquals("productList.size is wrong", 3, productList.size());
}
}
```

在每个单元测试方法中，我们使用了 JUnit 框架所提供的 `Assert.assertXxx()` 断言方法来验证期望值与实际值是否一致，这正是单元测试的基本做法。虽然 JUnit 提供了良好的单元测试框架，但似乎调用它所提供的断言方法却不太方便。比如在每个断言方法中都需要提供一个断言失败消息，实际上每种断言方法就能够代表每种测试的具体含义，似乎这里的断言失败消息有些多余。此外，期望值与实际值到底谁在前谁在后，我们不小心放颠倒了就容易犯下严重错误。

其实 Spring Boot 测试框架已经为我们考虑了以上这些问题，它依赖于 AssertJ 类库，它的强项就在于做断言这件事情，正好补充了 JUnit 框架在断言方面的不足之处。

AssertJ: <https://joel-costigliola.github.io/assertj/index.html>。

我们可轻松地将 JUnit 断言改为 AssertJ 断言，如表 6-1 所示。

表 6-1 JUnit 断言改为 AssertJ 断言

JUnit 断言	AssertJ 断言
<code>Assert.assertNotNull("product is null", product);</code>	<code>Assertions.assertThat(product).isNotNull();</code>

续表

JUnit 断言	AssertJ 断言
<code>Assert.assertEquals("product.price is wrong", 9000, product.getPrice());</code>	<code>Assertions.assertThat(product.getPrice()).isEqualTo(9000);</code>
<code>Assert.assertTrue("delete failure", result);</code>	<code>Assertions.assertThat(result).isTrue();</code>
<code>Assert.assertEquals("productList.size is wrong", 3, productList.size());</code>	<code>Assertions.assertThat(productList).hasSize(3);</code>

可见，使用 AssertJ 断言后，我们无须提供断言失败消息。此外，我们使用了严格的调用顺序来完成断言过程，首先通过调用 `assertThat()` 方法来传入期望值，随后再通过不同的断言方法来传入实际值（例如，`isEqualTo(9000)`），或者直接做出相应的断言操作（例如，`isNotNull()`），整个断言调用过程是一个流式的。

运行 `ProductServiceTest` 单元测试类后，我们一定会在运行结果中看到所有的单元测试都是失败的，如图 6-1 所示。

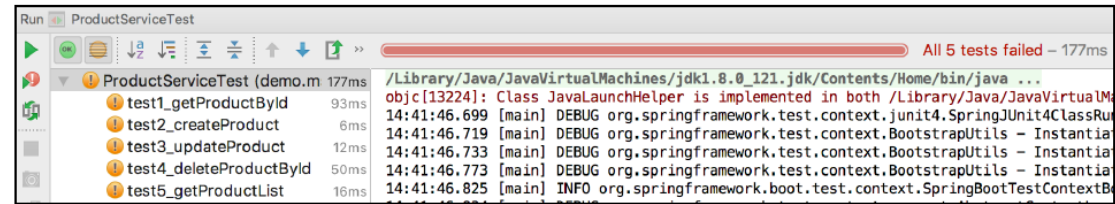


图 6-1 单元测试全部失败

此时出现单元测试全部失败的现象是正常的，因为待测对象 `ProductService` 类目前仍未做任何实现，因此调用其中每个方法都是失败的。

需要注意的是，运行 Spring Boot 单元测试时，无须运行 `ProductApplication` 应用程序，因为单元测试框架会自动运行该程序。此外，要做到以上批量运行单元测试的效果，需要先将 IntelliJ IDEA 中的光标移动到方法外部，再运行 `ProductServiceTest` 程序。

我们接下来要做的就是让以上单元测试结果变为全部通过，可以逐个实现 `ProductService` 类的每个方法，逐个运行相应的单元测试方法，逐个将失败变为成功，这正是 TDD 给我们所带来的乐趣，让我们更加方便地以用户的视角进行开发。

当然，如果有把握的话，我们也可一次性完成 `ProductService` 的所有实现，以下便是它最终实现的代码。

```
package demo.msa.product.service;
```

```
import demo.msa.product.model.Product;
import org.springframework.stereotype.Service;
import org.springframework.util.ReflectionUtils;

import java.lang.reflect.Field;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.concurrent.atomic.AtomicInteger;

@Service
public class ProductService {

    private final static List<Product> productList = new ArrayList<>();

    private final static AtomicInteger idGenerator = new AtomicInteger(1);

    static {
        productList.add(new Product(generateId(), "MacBook", 10000,
getcurrentTime()));
        productList.add(new Product(generateId(), "MacBook Air", 7000,
getcurrentTime()));
        productList.add(new Product(generateId(), "MacBook Pro", 12000,
getcurrentTime()));
    }

    private static int generateId() {
        return idGenerator.getAndIncrement();
    }

    private static long getcurrentTime() {
        return System.currentTimeMillis();
    }

    public Product getProductById(long id) {
        for (Product product : productList) {
```

```
        if (product.getId() == id) {
            return product;
        }
    }
    return null;
}

public Product createProduct(String name, int price) {
    Product product = new Product(generateId(), name, price, getCurrentTime());
    boolean result = productList.add(product);
    if (result) {
        return product;
    }
    return null;
}

public Product updateProduct(long id, Map<String, Object> fieldMap) {
    Product product = getProductById(id);
    if (product != null) {
        for (Map.Entry<String, Object> fieldEntry : fieldMap.entrySet()) {
            Object fieldValue = fieldEntry.getValue();
            if (fieldValue != null) {
                String fieldName = fieldEntry.getKey();
                Field field = ReflectionUtils.findField(Product.class, fieldName);
                ReflectionUtils.makeAccessible(field);
                ReflectionUtils.setField(field, product, fieldValue);
            }
        }
        return product;
    }
    return null;
}

public boolean deleteProductById(long id) {
    Iterator<Product> products = productList.iterator();
    while (products.hasNext()) {
        Product product = products.next();
        if (product.getId() == id) {
```

```

        products.remove();
        return true;
    }
}
return false;
}

public List<Product> getProductList() {
    return productList;
}
}

```

当然，以上实现只是一个示例版本，实际版本往往不会是这样的，而是通过操作数据库来完成相应业务逻辑。

当我们再次运行 `ProductServiceTest` 程序时，将看到所有的单元测试都变为通过，如图 6-2 所示。

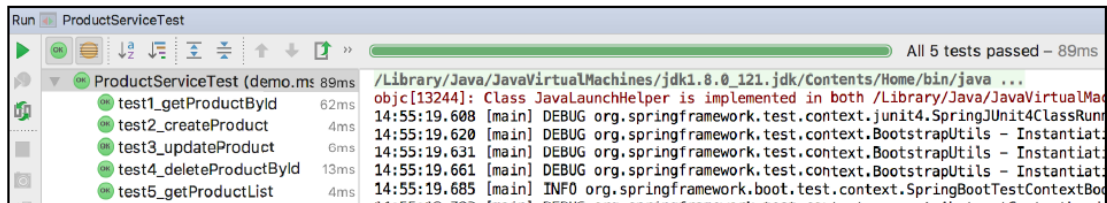


图 6-2 单元测试全部通过

以上便是 TDD 的整个过程，我们不再是先完成开发，再进行测试，而是测试与开发交替进行，并站在调用者的角度上，对需要完成的代码进行更加有针对性的测试，开发过程更加专业，也更加高效。大家可在平时工作中亲自实践这种开发模式，一定会有更加深刻的体会。

通过以上对 Service 层的单元测试，我们确保了 Service 层并无明显漏洞，可以放心地交给 Controller 层来调用了。下面我们将针对 Controller 层对外暴露的 REST API 进行单元测试，希望接下来的单元测试也能顺利完成。

6.1.3 测试 REST API

我们同样使用 Spring Boot 测试框架来搭建 `ProductController` 的单元测试类 `ProductControllerTest`，与 `ProductServiceTest` 类相比，在相似中却存在着差异。

```
package demo.msa.product.test;

import org.junit.FixMethodOrder;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.MethodSorters;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@FixMethodOrder(MethodSorters.NAME_ASCENDING)
public class ProductControllerTest {

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    public void test1_getProductById() throws Exception {
    }

    @Test
    public void test2_createProduct() throws Exception {
    }

    @Test
    public void test3_updateProduct() throws Exception {
    }

    @Test
    public void test4_deleteProductById() throws Exception {
    }

    @Test
    public void test5_getProductList() throws Exception {
    }
}
```


与 `ProductServiceTest` 不同的是,此时我们在 `@SpringBootTest` 注解中使用了 `webEnvironment` 属性,并将其设置为 `SpringBootTest.WebEnvironment.RANDOM_PORT`,表示运行 `ProductControllerTest` 将启动一个 Web 环境,同时对外暴露一个随机的端口号(而不是默认的 8080 端口号)。此外,我们使用 `@Autowired` 注解在 `ProductControllerTest` 类中注入了 `TestRestTemplate` 对象,该对象是对 `RestTemplate` 对象的封装,仅在测试环境下使用,可通过该对象来发送 REST 请求并获取相应的 JSON 响应。

接下来,我们将逐一对 `ProductController` 类中各个 REST API 进行单元测试。

以下是测试 `GET:/product/{id}` 的单元测试代码,我们通过 `TestRestTemplate` 对象来发送 REST 请求,并通过 `AssertJ` 所提供的 `Assertions` 对象来完成断言验证。

```
...
import demo.msa.product.model.Product;
import org.assertj.core.api.Assertions;
...
@Test
public void test1_getProductById() throws Exception {
    Product product = restTemplate.getForObject("/product/1", Product.class);
    Assertions.assertThat(product.getId()).isEqualTo(1);
    Assertions.assertThat(product.getName()).isEqualTo("MacBook");
    Assertions.assertThat(product.getPrice()).isEqualTo(10000);
}
...
```

以上方式虽然可行,但如果 `Product` 对象中包含其他更多的属性,那么我们就要编写大量的断言语句。如果能将调用 REST API 返回的 JSON 字符串做一个校验,是否能简化我们的断言过程呢?我们不妨将以上单元测试代码进行如下改写。

```
...
@Test
public void test1_getProductById() throws Exception {
    String expected = "\"id\":1,\"name\":\"MacBook\",\"price\":10000";
    String actual = restTemplate.getForObject("/product/1", String.class);
    Assertions.assertThat(actual).contains(expected);
}
...
```

此时我们只需比较实际值(actual)中是否包含期望值(expected),就能断言单元测试是否

通过。需要注意的是，此时的 `expected` 并非是一个完整的 JSON 字符串，其中头尾的花括号被我们去掉了，如果不去掉则无法通过 `contains` 断言（大家可以思考一下为什么）。可见，这种方式虽然可行，但对于复杂的 JSON 结构而言，似乎显得比之前的方案更为麻烦。因此我们想到了另一种做法，通过解析 JSON 对象，并针对关键性属性进行逐个断言。

我们可通过 `JsonPath` 技术来实现对 JSON 对象的解析，实际上 `JsonPath` 是一种类似于 `XPath` 的规范，在 Java 中我们可使用 `Jayway` 开源的 `JsonPath` 实现来完成 JSON 对象的解析工作。

`JsonPath` 规范: <http://goessner.net/articles/JsonPath/>。

`Jayway JsonPath`: <https://github.com/json-path/JsonPath>。

此外，`Jayway` 还提供了一个在线的 `JsonPath` 工具（`Jayway JsonPath Evaluator`），当 JSON 字符串较为复杂时，我们可使用该工具来提高工作效率。

`Jayway JsonPath Evaluator`: <http://jsonpath.herokuapp.com/>。

以下是使用了 `JsonPath` 改写后的单元测试代码，我们可分别针对需要断言属性加以控制。

```
...
import com.jayway.jsonpath.DocumentContext;
import com.jayway.jsonpath.JsonPath;
...
@Test
public void test1_getProductById() throws Exception {
    String json = restTemplate.getForObject("/product/1", String.class);
    DocumentContext ctx = JsonPath.parse(json);
    Assertions.assertThat(ctx.read("$.id")).isEqualTo(1);
    Assertions.assertThat(ctx.read("$.name")).isEqualTo("MacBook");
    Assertions.assertThat(ctx.read("$.price")).isEqualTo(10000);
}
...
```

此外，还有一种叫 `JSONassert` 的类库也能断言 JSON 字符串，这种方式上面提到的比较字符串方法相类似，只是增加了更为特殊的效果。

`JSONassert`: <http://jsonassert.skyscreamer.org/>。

使用 `JSONassert`，我们可按照非严格的方式对 JSON 字符串进行断言，可见这种方式显得更加灵活。

```

...
import org.skyscreamer.jsonassert.JSONAssert;
...

@Test
public void test1_getProductById() throws Exception {
    String expected = "{\"id\":1,\"name\":\"MacBook\",\"price\":10000}";
    String actual = restTemplate.getForObject("/product/1", String.class);
    JSONAssert.assertEquals(expected, actual, false);
}
...

```

以上我们讨论了 4 种断言 JSON 数据的方法，到底应该用哪种？大家不妨根据实际情况来灵活选择。

比如，我们可通过 JSONassert 来断言 POST:/product 请求后所返回的 JSON 字符串。

```

...
@Test
public void test2_createProduct() throws Exception {
    ProductRequest request = new ProductRequest();
    request.setName("iMac");
    request.setPrice(8000);

    String expected = "{\"id\":4,\"name\":\"iMac\",\"price\":8000}";
    String actual = restTemplate.postForObject("/product", request,
String.class);
    JSONAssert.assertEquals(expected, actual, false);
}
...

```

当然，也能混合使用 JSONassert 与 AssertJ 进行断言。

```

...
import org.springframework.http.HttpMethod;
import org.springframework.http.ResponseEntity;
...
@Test
public void test3_updateProduct() throws Exception {

```

```

        String expected = "{\"id\":4,\"name\":\"iMac\",\"price\":9000}";
        Map<String, Object> fieldMap = new HashMap<>();
        fieldMap.put("price", 9000);
        RequestEntity<Map<String, Object>> requestEntity = new RequestEntity<>
(fieldMap, HttpMethod.PUT, new URI("/product/4"));
        String actual = restTemplate.exchange(requestEntity, String.class).
getBody();
        JSONAssert.assertEquals(expected, actual, false);

        int price = JsonPath.read(actual, "$.price");
        Assertions.assertThat(price).isEqualTo(9000);
    }
    ...

```

对于简单的情况而言，我们都能使用 JSONAssert 来实现断言。

```

    ...
    @Test
    public void test4_deleteProductById() throws Exception {
        String expected = "{\"id\":4,\"name\":\"iMac\",\"price\":9000}";
        String actual = restTemplate.exchange("/product/4", HttpMethod.DELETE,
null, String.class).getBody();
        JSONAssert.assertEquals(expected, actual, false);
    }
    ...

```

但对于相对复杂的 JSON 字符串而言，虽然 JSONAssert 仍然可以处理，但似乎感觉上还是繁重。

```

    ...
    @Test
    public void test5_getProductList() throws Exception {
        String expected = "{\"items\":[{\"id\":1,\"name\":\"MacBook\",
\"price\":10000},{\"id\":2,\"name\":\"MacBook Air\",\"price\":7000},{\"id\":3,
\"name\":\"MacBook Pro\",\"price\":12000}],\"total\":3}";
        String actual = restTemplate.getForObject("/product", String.class);
        JSONAssert.assertEquals(expected, actual, false);
    }
    ...

```

有时候，我们也会返璞归真，回归最初的原点。

```
...
import demo.msa.product.response.ProductResponse;
...
@Test
public void test5_getProductList() throws Exception {
    ProductResponse response = restTemplate.getForObject("/product",
ProductResponse.class);
    Assertions.assertThat(response.getProductList()).isNotNull();
    Assertions.assertThat(response.getTotal()).isEqualTo(3);
}
...
```

不论哪种断言方式，我们都能完成同样的目标。但对于开发团队而言，最好还是选用其中一种方式作为团队写单元测试的代码规范。

当所有 REST API 都通过单元测试后，我们接下来要做的是让 REST API 测试变得自动化。可以想象，当一个 REST API 的实现细节发生变化后，如果能自动运行自动测试框架，对所有的可能影响到的地方都进行一遍回归测试，这样的效果将是多么好啊！只要 REST API 稳定了，面向用户的前端应用程序才能更加稳定。在下节中，我们将针对 REST API 的自动化测试做出一些尝试。

6.2 搭建 REST API 自动化测试框架

测试人员不仅需要确保整个应用系统的质量，还要确保底层 REST API 不出任何问题，实际上 REST API 才是整个应用系统的基础，它们稳定了，整个应用系统才能稳定。虽然我们编写的 JUnit 单元测试可在 Maven 构建中自动运行，甚至也能在 Jenkins 进行持续集成中自动地执行单元测试，这样也能完成 REST API 的自动化测试工作，但这种方式对于测试人员而言，却带来了一个不小的麻烦。由于 JUnit 测试属于“白盒测试”，哪怕测试代码是开发人员编写的，测试人员也要看得懂这些代码，才能更有效地执行测试用例，就更别说让测试人员自己去编写单元测试代码了，因此这项工作对于测试人员而言是有一定挑战的。我们更加提倡的做法是，除了开发人员的 JUnit 测试，还需要通过测试人员进行“黑盒测试”来保证 REST API 的质量，测试人员需要编写测试用例以及相关的测试脚本，从而在单元测试的基础上，再次验证 REST API 的可用性。经过探索后我们发现，Postman 不仅为开发人员提供了 REST 客户端调用工具，同时它也为测试人员提供了一个 REST API 的自动化测试框架。

我们现在就利用 Postman 来实现自动化测试，但我们仍然需要从手工测试开始。

6.2.1 使用 Postman 手工测试 REST API

Postman 对于开发人员而言并不陌生，它是一款好用的 REST 客户端工具，我们可通过它来调用 REST 请求，并获取相应的 JSON 响应。当然，这些只是 Postman 最基础用法，下面我们将对 Postman 在测试方面进行一些尝试。

Postman 官网：<https://www.getpostman.com/>。

在 Postman 左侧的面板中，可以看到两个选项卡：History（历史）与 Collections（集合）。历史用于记录每次发送请求的历史记录，点击后可查看曾经发生过的调用场景；集合用于管理需要调用的请求集合，也就是说，我们可将需要调用的请求放入相应的集合中。

大家可能会问：集合除了可以归类请求，难道就没有其他更有价值的用途吗？我们将在下文中为大家揭晓。

现在我们要做的第一件事情是创建一个集合，它的名称为“Product REST API”，该集合用于管理 Product 服务相关的 REST API。

我们可在 New Tab 中创建一个请求，在请求路径中输入“{HYPERLINK "http://localhost:8080/product/1"}”，随后可点击 Send 按钮，此时将发送该请求，随后将在下方的 Body 选项卡中看到输出的 JSON 数据，如图 6-3 所示。

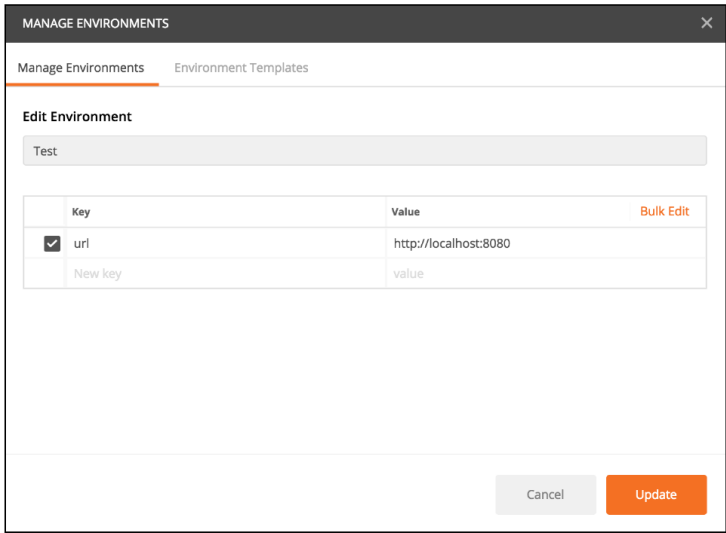


图 6-3 发送 REST 请求

我们也能点击 Send 按钮旁边的 Save 按钮，可将该请求保存起来，并将它命名为“create_product”，同时还能将它加入“Product REST API”这个集合中。

需要注意的是，在调用请求之前，我们需要首先启动服务端应用程序，否则请求将无法得到成功响应。

同样，我们也能用以上方法来创建并保存其他种类请求。当请求种类较多时，如果都放在同一集合中，难免会造成管理上的麻烦，我们需要将其进行分类。幸运的是，Postman 已经为我们提供了一个叫 Folder（文件夹）的东西，每个集合中可创建多个文件夹，每个文件夹中可包括多个请求，这是一个非常清晰的三层结构。

然而，在“集合—文件夹—请求”这样的三层结构中，我们需要进行测试的正是第一级集合，当然也能根据指定的文件夹进行测试。这样的测试框架非常灵活，那么具体应该如何测试 REST API 呢？

在正式使用 Postman 测试 REST API 之前，我们首先要掌握的是 Postman 提供的两种 key-value 变量的使用场景。

（1）Environments（环境变量）：在某个集合范围内有效。

（2）Globals（全局变量）：在所有集合范围内共享。

对于“http://localhost:8080/product/1”这个请求地址而言，我们可以将其中的 URL（http://localhost:8080）放入环境变量中进行管理，如图 6-4 所示。

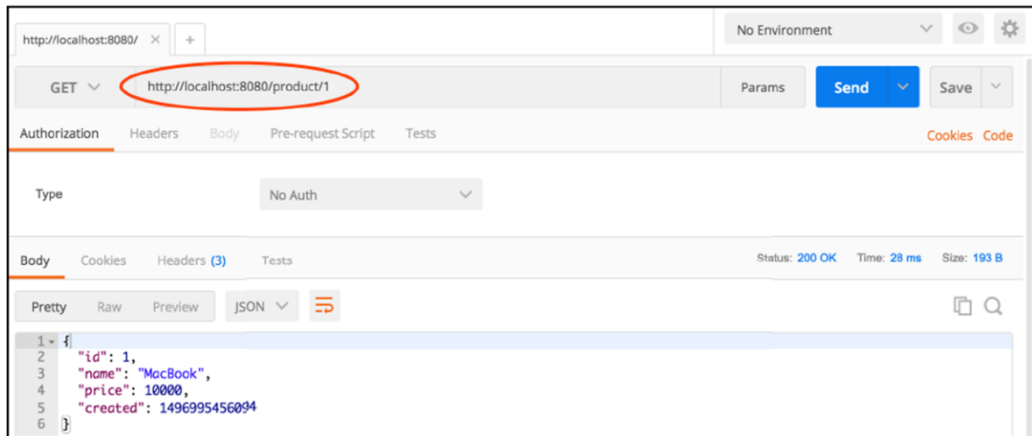


图 6-4 创建环境变量

随后我们就可将请求路径替换为“{{url}}/product/1”，同时在右上角选择对应环境变量“Test”，继续点击 Send 按钮，随后将看到同样的 JSON 输出，如图 6-5 所示。

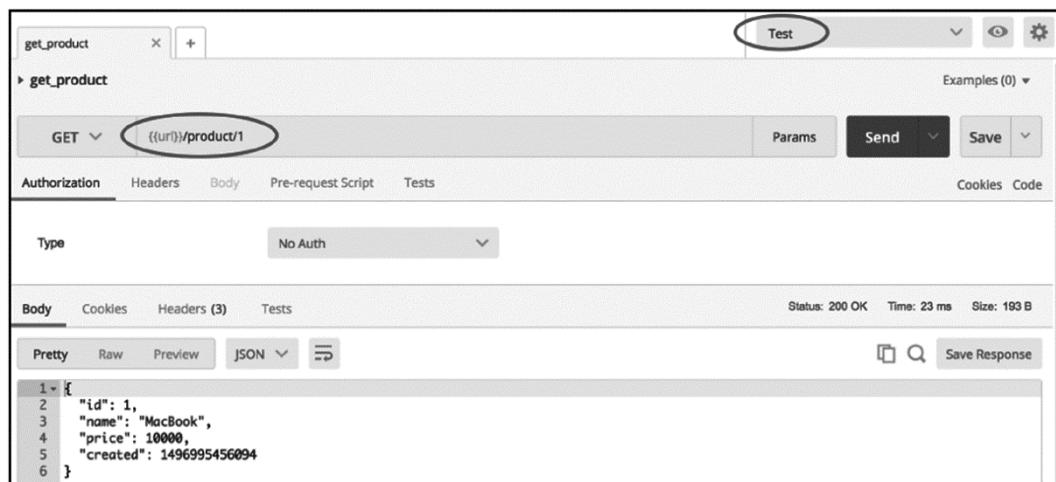


图 6-5 使用环境变量

我们可以在请求地址栏下方的 **Pre-request Script** 与 **Tests** 选项卡中输入一些测试脚本，这两处脚本分别具有不同的用途。

(1) **Pre-request Script**（前置脚本）：在发送请求前执行。

(2) **Tests**（后置脚本）：在发送请求后执行。

在前置脚本中，我们可操作一些环境变量或全局变量。

```
// 设置一个环境变量
postman.setEnvironmentVariable("variable_key", "variable_value");

// 清除一个环境变量
postman.clearEnvironmentVariable("variable_key");

// 设置一个全局变量
postman.setGlobalVariable("variable_key", "variable_value");

// 清除一个全局变量
postman.clearGlobalVariable("variable_key");
```

在后置脚本中，我们不仅可以进行以上操作，还能做更多的事情。

```
// 测试响应体中是否包含指定字符串
tests["Body matches string"] = responseBody.has("string_you_want_to_search");
```



```
// 测试响应体是否等于指定字符串
tests["Body is correct"] = responseBody === "response_body_string";

// 测试 JSON 对象中的相关属性
var jsonData = JSON.parse(responseBody);
tests["Your test name"] = jsonData.value === 100;

// 将响应体从 XML 字符串转换为 JSON 对象
var jsonObject = xml2Json(responseBody);

// 测试 Content-Type 响应头中是否存在
tests["Content-Type is present"] = postman.getResponseHeader("Content-Type");

// 测试响应时间是否小于 200 毫秒
tests["Response time is less than 200ms"] = responseTime < 200;

// 测试状态码是否等于 200
tests["Status code is 200"] = responseCode.code === 200;

// 测试 POST 请求是否成功
tests["Successful POST request"] = responseCode.code === 201 || responseCode.code === 202;

// 测试状态码名称中是否包含指定字符串
tests["Status code name has string"] = responseCode.name.has("Created");

// 测试 JSON Schema
var schema = {
  "items": {
    "type": "boolean"
  }
};
var data1 = [true, false];
var data2 = [true, 123];
tests["Valid Data1"] = tv4.validate(data1, schema);
tests["Valid Data2"] = tv4.validate(data2, schema);
console.log("Validation failed: ", tv4.error);
```

实际上，Postman 内置了许多工具库，例如上面提到的“测试 JSON Schema”，我们用到的 tv4 对象其实是 Postman 内置的 Tiny Validator 类库所提供的，它用于校验 JSON 数据的结构。

Tiny Validator: <http://geraintluff.github.io/tv4/>

当然还有更多的内置工具库，例如，Lodash、cheerio、SugarJS 等，我们可在 Postman 官方文档中了解更多相关信息，这里我们就不再展开讲解了。

对于目前的 GET:/product/{id} 这个 REST API 而言，我们可对响应状态码与实际返回的 JSON 数据进行校验。

```
tests["Status code is 200"] = responseCode.code == 200;

var json = JSON.parse(responseBody);
tests["Check id"] = json.id == 1;
tests["Check name"] = json.name == "MacBook";
tests["Check price"] = json.price == 10000;
```

将以上脚本放入 Tests 选项卡中，再次点击 Send 按钮，在下方的 Tests 选项卡中能看到所有的测试全部通过，如图 6-6 所示。

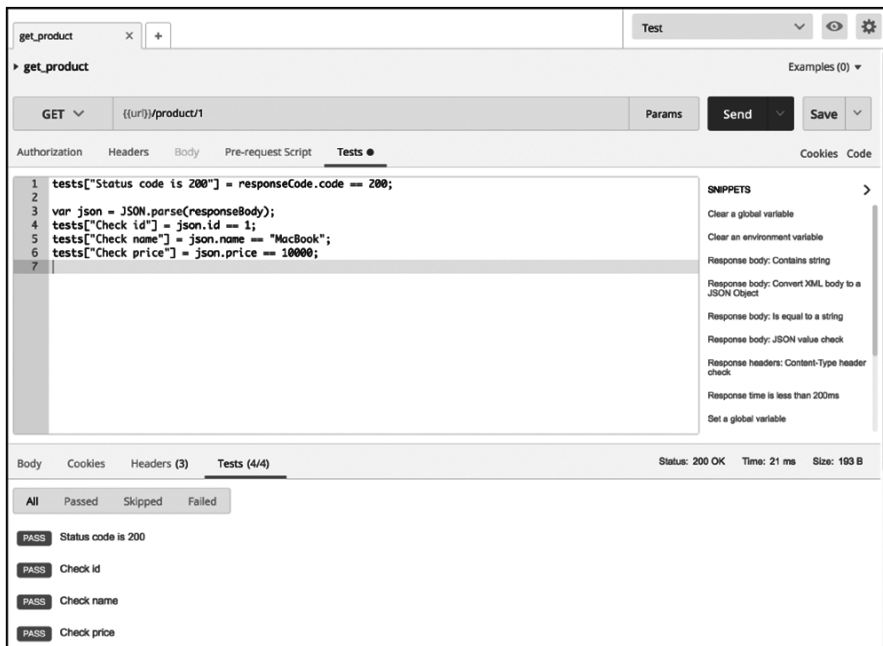


图 6-6 执行测试脚本

以上我们所做的只是针对 ID 为 1 情况下的测试用例，如果 ID 为其他情况时，我们要么改写现在的测试用例，要么再创建其他测试用例。似乎这两种方法并非最佳解决方案，其实我们更需要的是一个测试用例“模板”，通过传入不同的 ID 参数到这个测试模版中，Postman 可根据测试模版分别执行测试用例，并依次校验相应的测试结果。振奋人心的是，Postman 已经为我们做到了这一切。

我们将请求地址改写为 `{{url}}/product/{{id}}`，其中的 `{{id}}` 是一个占位符，可从外部数据文件中读取。此外，我们也将 Tests 中的测试脚本做出以下调整。

```
tests["Status code is 200"] = responseCode.code == 200;

var json = JSON.parse(responseBody);
tests["Check id"] = json.id == data.id;
tests["Check name"] = json.name == data.name;
tests["Check price"] = json.price == data.price;
```

测试脚本中的 `data` 对象同样也从外部数据文件中读取，以前具体的测试用例现在已变为抽象的测试模板，如图 6-7 所示。

此时我们不能点击 Send 按钮来执行测试用例，而是准备一个 CSV 文件（文件名可任意，例如，`get_product.csv`），将测试用例所需的参数输入到该文件中，如图 6-8 所示。

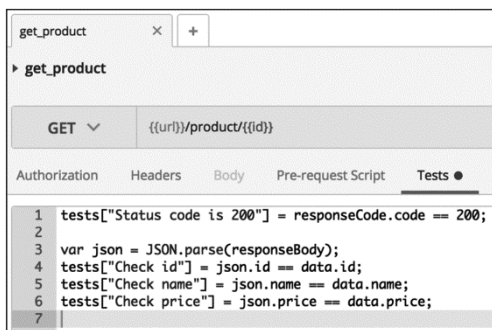


图 6-7 编写测试模板

	A	B	C
1	id	name	price
2	1	MacBook	10000
3	2	MacBook Air	7000

图 6-8 准备测试参数

以上 CSV 文件中可使用 Excel 来编辑，其中包括 id、name、price 三列，我们在请求地址中使用的 `{{id}}` 占位符就对应 CSV 文件中的 id 列，在测试脚本中使用的 `data` 变量就映射于 CSV 文件中的某一行。因此我们可使用 `data.id`、`data.name`、`data.price` 等语法来获取 CSV 文件中的相关数据，其实 `{{id}}` 占位符中的数据也是通过 `data.id` 来获取的。

除了 CSV 格式的数据文件，Postman 还支持 JSON 格式的数据文件，可能不如 CSV 文件那么容易编写和维护。

```
[{
  "id": 1,
  "name": "MacBook",
  "price": 10000
}, {
  "id": 2,
  "name": "MacBook Air",
  "price": 7000
}]
```

接下来我们要做的是，点击 Postman 左上角的 Runner 按钮，此时将打开一个叫 Collection Runner 的窗口。首先选择“Get Product”这个集合的文件夹（表示测试目标），然后在 Environment 下拉框中选择对应的环境变量，并将 Iterations 设置为 2（设置测试的迭代次数），最后将以上 CSV 文件进行上传，如图 6-9 所示。

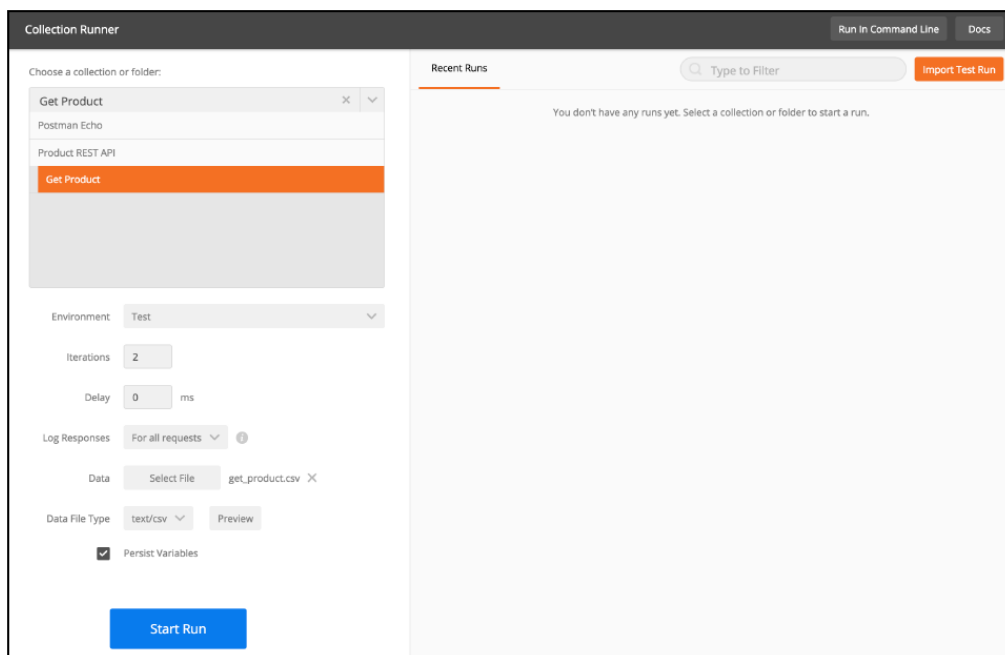


图 6-9 配置测试环境

点击 Start Run 按钮，此时开始读取数据文件并迭代执行对应的测试用例，最终的测试报告将输出到下一界面中，如图 6-10 所示。

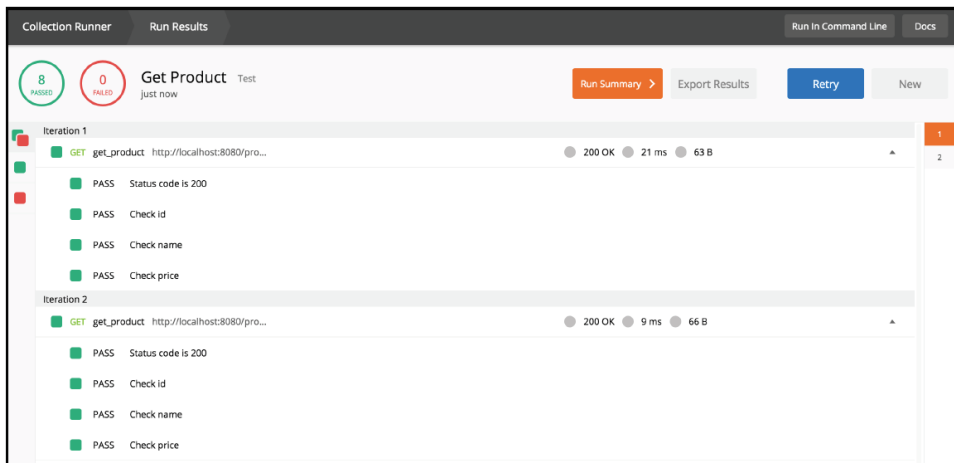


图 6-10 输出测试结果

我们也能点击每个迭代中的请求链接，此时可查看具体的请求与响应，如图 6-11 所示。



图 6-11 查看请求与响应

当然，我们也能开启 Postman 的控制台（点击 View / Show Postman Console 菜单），更加直观地查看整个请求与响应的相关数据，如图 6-12 所示。

需要补充说明的是，我们在测试脚本中也能调用 `console.log(...)` 方法来输出需要调试的信息，这些调试信息也将输出到 Postman 控制台中。此外，我们还能利用 Postman 提供的 DevTools 完成更为强大的功能（点击 View / Show DevTools 菜单），它的用法与 Chrome 中提供的开发者工具相同。

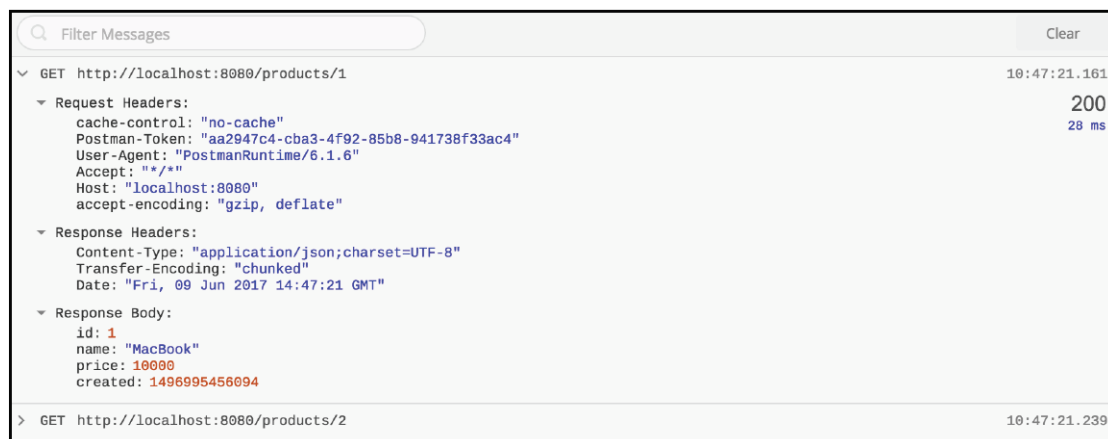


图 6-12 Postman 控制台

实际上我们目前只是针对每个 REST API 编写了相应的测试用例，并验证了这些测试用例的可测性，这个过程是通过手工来完成的。但是，我们无法做到批量执行大量的测试用例集合，如果单纯依靠手工来完成测试，工作效率方面是很难得到提高的。实际上，Postman 官方已经为我们考虑了这个问题，并提供了一款叫 Newman 的命令行工具，我们可通过它来批量测试 REST API。

6.2.2 使用 Newman 批量测试 REST API

Newman 是基于 Node.js 开发的命令行应用程序，我们可通过命令行的方式高效地运行在 Postman 中制作的测试用例，并在控制台中看到所生成的测试报告，此外还能生成其他类型的测试报告。

我们可在 NPM 站点上了解 Newman 的具体使用方法。

Newman NPM: <https://www.npmjs.com/package/newman>。

首先，我们需要通过 NPM 命令并以全局的方式将其安装到命令行中。

```
npm install newman --global;
```

随后，可通过 `newman run` 命令来运行基于 Postman 的测试用例。

```
newman run collection.json \  
--environment environment.json \  
--iteration-data get_product.csv \  

```

```
--iteration-count 2 \
--folder "Get Product" \
--reporters cli,json,html,junit
```

在 `newman run` 命令中，我们需要提供一份从 Postman 中导出的集合文件（例如，`collection.json`），还需要提供以下相关选项。

（1）`--environment`：用于设置集合对应的环境变量文件，可从 Postman 中导出（例如，`environment.json`）。

（2）`--iteration-data`：用于设置测试所需的迭代数据文件，可从 Postman 中导出（例如，`get_product.csv`）。

（3）`--iteration-count`：用于设置测试的迭代次数，可根据迭代数据文件中的行数来决定。

（4）`--folder`：用于设置 Postman 集合中所对应的文件夹，如果不指定，那么将测试整个集合。

（5）`--reporters`：用于设置测试报告格式，可使用四种测试报告——`cli`、`json`、`html`、`junit`。

其中 `cli` 是默认的测试报告格式，用于在命令行中直接输出，如图 6-13 所示。

Product REST API		
Iteration 1/2		
Get Product		
get_product		
GET http://localhost:8080/products/1 [200 OK, 193B, 50ms]		
✓ Status code is 200		
✓ Check id		
✓ Check name		
✓ Check price		
Iteration 2/2		
get_product		
GET http://localhost:8080/products/2 [200 OK, 196B, 7ms]		
✓ Status code is 200		
✓ Check id		
✓ Check name		
✓ Check price		
	executed	failed
iterations	2	0
requests	2	0
test-scripts	2	0
prerequest-scripts	0	0
assertions	8	0
total run duration: 146ms		
total data received: 129B (approx)		
average response time: 28ms		

图 6-13 Newman 生成的命令行测试报告

此外，也将当前目录下的 `newman` 子目录中生成其他类型的测试报告，如图 6-14 所示。



图 6-14 Newman 生成的其他测试报告

虽然，使用 Newman 可批量执行 Postman 中的测试用例，但还是无法通过自动化的方式来工作。我们接下来想做的是，通过一个 Jenkins 来调用 Newman，从而搭建一个 REST API 的自动化测试框架。

6.2.3 搭建 REST API 自动化测试框架

在 Jenkins 中，我们首先需要构建一个自由风格的软件项目，如图 6-15 所示。

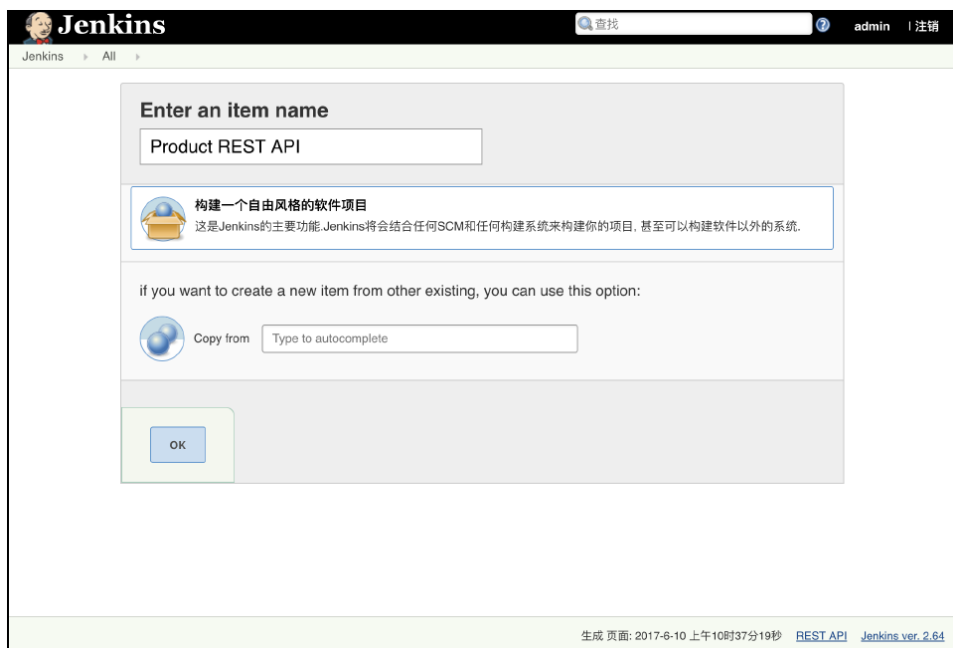


图 6-15 构建 Jenkins 项目

接下来，在构建中的 `Execute shell` 中输入以下 Newman 命令，输入后的效果如图 6-16 所示。

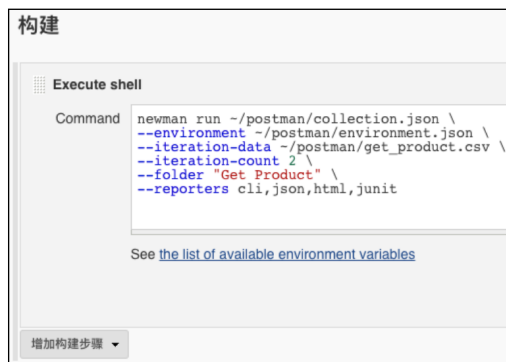


图 6-16 输入 Newman 命令

在以上命令中，我们使用了绝对路径来定位 `newman run` 命令所需的文件。当然，这里也能添加更多的命令行脚本来执行其他测试用例。

如果想要查看生成的 JUnit 格式的测试报告，我们可以在构建后操作中的 Publish JUnit test result report 中输入 JUnit 测试报告的 XML 文件路径，在路径中支持通配符，如图 6-17 所示。



图 6-17 集成 JUnit 测试报告

最后，我们需要在构建触发器中的 Build periodically 中输入一个 Cron 表达式，作为定时测试周期，如图 6-18 所示。



图 6-18 设置定时测试周期

至此，一款 REST API 自动化测试框架已基本搭建完毕，开发人员无须编写任何 JUnit 单元测试代码，只需测试人员通过 Postman、Newman、Jenkins 等工具就能完成整个自动化测试工作，节省了开发人员的时间，也提高了测试人员的效率。

实际上，REST API 是由后端开发人员使用 Spring Boot 框架所开发的，并通过 JUnit 框架确保了它的可用性，随后测试工程师可通过 Postman 来创建相关的测试用例，并通过 Jenkins + Newman 的方式来自动化执行测试用例。这个过程看似比较顺利，但实际上却存在一个工作效率上的问题。测试人员必须等待后端开发人员将 REST API 发布出来以后，才能编写相应的测试用例。此外，前端开发人员也必须等待后端人员将 REST API 发布以后，才能完成前端页面的数据渲染工作。可见，测试人员与前端开发人员都需要等待后端开发人员的工作成果，这样的流程严重影响了开发与测试效率，如何改进呢？我们只需自动生成 REST API 文档，就能轻松解决这个问题。

6.3 自动生成 REST API 文档

作为后端开发人员，在进入微服务开发之前，我们首先需要做的是将微服务对外提供的 REST API 进行文档化。这样做不仅有利于前端开发人员能够基于该文档来仿制前端所需的数据，还能帮助测试人员更早地编写接口测试用例与自动化测试脚本。可见，完成 REST API 文档化是我们首要的任务，它能让三方（后端、前端、测试）并行工作，显著提高开发效率。如果 REST API 文档能够尽早发布出来，将使整个并行工作更加提前开始，从而为我们后续的工作节省了宝贵的时间。因此，我们有必要让 REST API 文档的编写过程做到自动化，以最高效的方式让 REST API 文档自动生成。

Swagger 是我们需要研究的第一款自动生成 REST API 文档的工具，下面我们就一起进入它的世界。

6.3.1 使用 Swagger 生成 REST API 文档

Swagger 是业界最流行的 API 开发框架，它的影响范围甚至跨越了整个 API 生命周期，从设计与文档到测试与部署。早在 2011 年，Swagger 官方就推出了一套 API 文档规范（Swagger RESTful API Documentation Specification），后来它成为了业界公认的 OpenAPI 文档规范（OpenAPI Specification）。3 年后，Swagger 文档规范发布了 2.0 版本，新版本让 REST API 文档规范变得更加成熟和精炼。

关于 Swagger 的更多细节以及 OpenAPI 文档规范，可从它的官网中获知，如图 6-19 所示。

Swagger 官网：<http://swagger.io/>。



图 6-19 Swagger 官网

通过对官网的学习，我们了解到 Swagger 官方推出了以下三款开源工具。

- (1) Swagger Editor: 用于编辑 REST API 文档。
- (2) Swagger Codegen: 用于生成 REST API 代码。
- (3) Swagger UI: 用于查看 REST API 文档。

我们可使用 Swagger Editor 并通过在线的方式来编辑 REST API 文档，如图 6-20 所示。

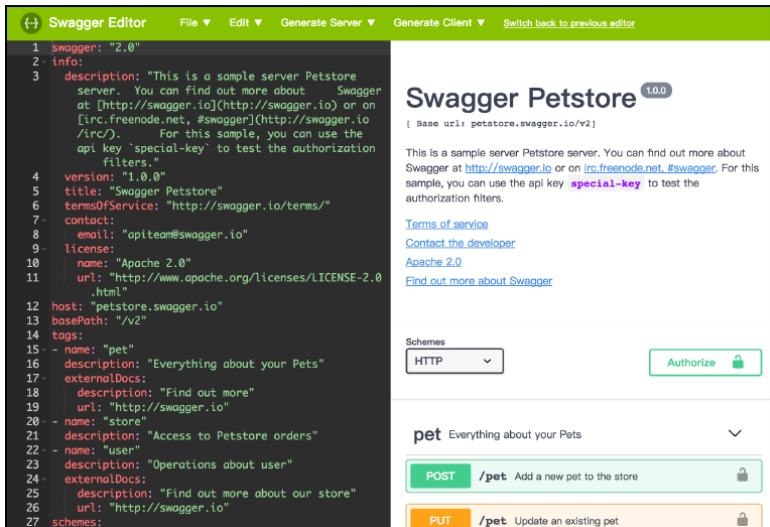


图 6-20 使用 Swagger Editor 编辑 REST API 文档

就像 Markdown 编辑器一样，我们可在左侧编写 Swagger 源文档（YAML 或 JSON 格式），此时将在右侧实时显示 REST API 文档（HTML 格式），还能在页面上进行一些交互。使用 Swagger Editor 编写 REST API 文档是一种畅快的体验，它真正做到了文档的“可见即所得”。

这么好的工具只能在线体验，大家一定会觉得有些遗憾，很想将它“据为己有”吧？兴奋的是，Swagger 官方已经为我们准备了 Swagger Editor 的 Docker 镜像，我们自己也能亲手搭建一个与官方同样的 Swagger Editor，只需在终端输入以下 Docker 命令即可。

```
docker run -d -p 8081:8080 swaggerapi/swagger-editor
```

当容器启动后，我们在浏览器中访问 <http://localhost:8081>，就能进入我们自己搭建的 Swagger Editor。

不仅如此，我们还能通过 Swagger 源文档来生成指定编程语言的服务端框架与客户端 SDK，这一切归功于 Swagger Codegen。

如果我们使用 Mac 操作系统，可以使用以下 Homebrew 命令来安装 Swagger Codegen。

```
brew install swagger-codegen
```

当我们准备好 Swagger 源文档以后，就能通过类似以下 Swagger Codegen 来生成特定编程语言的代码框架。

```
swagger-codegen generate -i <Swagger 源文件地址> -l <编程语言>
```

其中，Swagger 源文件地址可以是本地路径，也可以是远程链接，编程语言可参考 Swagger Editor 中所给出的可选种类。

例如，我们可使用以下 Swagger Codegen 命令生成基于 Spring 的服务端框架。

```
swagger-codegen generate -i http://petstore.swagger.io/v2/swagger.json -l spring
```

还可以使用以下 Swagger Codegen 命令生成生成基于 Java 的客户端 SDK。

```
swagger-codegen generate -i http://petstore.swagger.io/v2/swagger.json -l java
```

我们先通过 Swagger Editor 来设计 REST API 文档，然后通过 Swagger Codegen 快速生成指定编程语言的服务端或客户端，我们称这种开发模式为“文档驱动开发”，它是目前比较新颖的敏捷开发模式。如果大家比较保守，仍然希望使用最经典的开发模式（先写代码，后出文档），Swagger 也能很好地帮助到我们。

对于 Java 而言，我们可在服务端中使用 Swagger Core，通过在源代码中使用 Java 注解的方

式来生成 Swagger 源文档。

Swagger Core: <https://github.com/swagger-api/swagger-core>。

此外，当服务端启动后，我们还能通过 Swagger UI 在浏览器中查看已发布的 REST API 文档。

对于其他编程语言，Swagger 官方与第三方也提供了与 Swagger Core 类似的文档生成工具。

当然，我们也能使用 Docker 在自己的服务器上运行 Swagger UI。

```
docker run -d -p 8082:8080 swaggerapi/swagger-ui
```

当容器启动后，我们在浏览器中访问 <http://localhost:8082>，就能进入我们自己搭建的 Swagger UI，如图 6-21 所示。

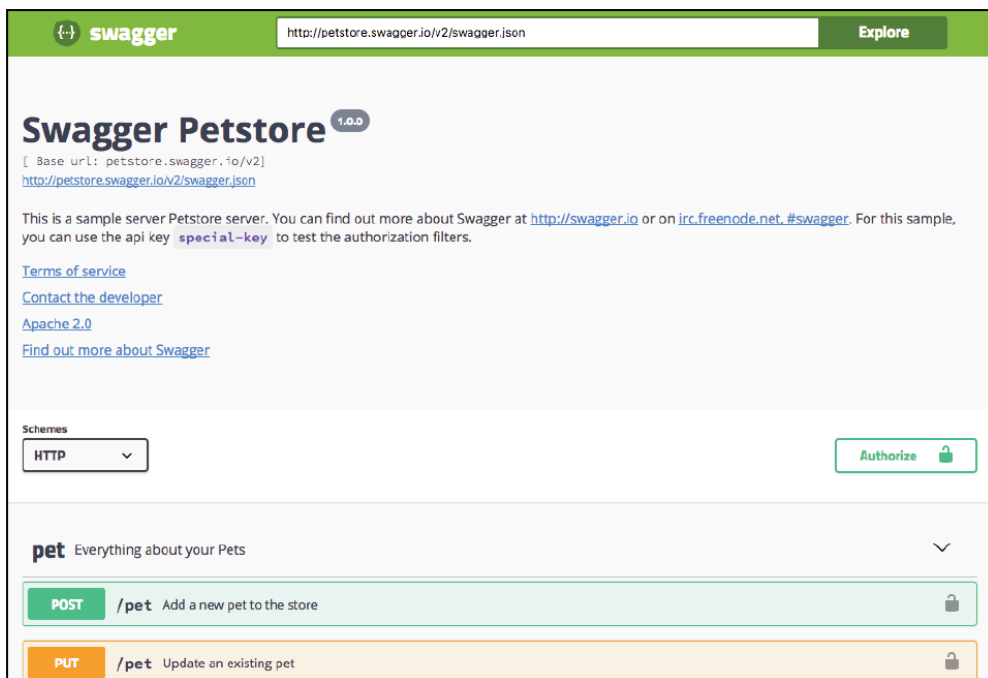


图 6-21 使用 Swagger UI 查看 REST API 文档

目前，Swagger Core 仅支持 JAX-RS 与 Servlet 实现的 REST API 的文档生成。但是，我们现在使用了 Spring Boot 来开发 REST API，此时的 Swagger Core 似乎变得无法使用了。

难道 Spring Boot 就真无法与 Swagger 整合吗？

这个问题被 Springfox 彻底解决了，它曾经是一个名为 swagger-springmvc 的开源项目，它能为基于 Spring MVC 框架的 Web 应用程序自动生成 REST API 文档。

Springfox: <https://springfox.github.io/springfox>。

我们接下来需要做的是，在 Spring Boot 项目的 pom.xml 配置文件中添加 Springfox 的 Maven 依赖。

```
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger2</artifactId>
  <version>2.7.0</version>
</dependency>
```

以上 springfox-swagger2 组件内部依赖于 Swagger Core 中的两个子模块：swagger-models 与 swagger-annotations，由于 Maven 依赖具备传递性，因此此时无须依赖 Swagger Core 的这两个子模块。

此外，我们还需添加一个经 Springfox 封装过的 Swagger UI 的 Maven 依赖。

```
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger-ui</artifactId>
  <version>2.7.0</version>
</dependency>
```

以上 springfox-swagger-ui 组件用于集成 Spring Web 框架，并提供 Swagger UI 形式的 REST API 文档。

Maven 依赖配置完毕后，我们就可在 Controller 方法上添加 Swagger 注解，定义 REST API 文档的所需的相关信息。

```
package demo.msa.product.controller;

...
import io.swagger.annotations.*;
...

@RestController
@RequestMapping(produces = "application/json")
```

```
public class ProductController {

    @ApiOperation("根据产品 ID 查询产品")
    @ApiImplicitParams({
        @ApiImplicitParam(value = "产品 ID", paramType = "path", dataType =
"long", name = "id", required = true)
    })
    @ApiResponses({
        @ApiResponse(code = 200, message = "成功"),
        @ApiResponse(code = 400, message = "错误请求"),
        @ApiResponse(code = 600, message = "无效的产品 ID")
    })
    @GetMapping("/product/{id}")
    public ResponseEntity<Product> getProductById(@PathVariable("id") long id) {
        // TODO: 尚未完成
        return ResponseEntity.badRequest().build();
    }

    ...
}
```

我们在 Controller 方法上使用了几种常用的 Swagger 注解，它们的作用各有不同。

- **@ApiOperation:** 用于定义 REST API 的摘要信息。
- **@ApiImplicitParams:** 用于定义 REST API 的请求参数，需要与 **@ApiImplicitParam** 注解组合使用。
- **@ApiImplicitParam:** 用于定义 REST API 参数的具体信息。
- **@ApiResponses:** 用于定义 REST API 的响应结果，需要与 **@ApiResponse** 注解组合使用。
- **@ApiResponse:** 用于定义 REST API 的状态码与状态消息，可使用自定义状态码。

需要注意的是，我们当前的目标只是通过 Swagger 注解来生成 REST API 文档，因此现在我们无须完成 Controller 方法。我们此时要做的是尽快生成 REST API 文档，当与前端开发人员进行沟通并确认后，我们将继续编写 Controller 方法中尚未完成的代码。

此时我们只是为一个 Controller 方法添加了 Swagger 注解，其他方法上也带有类似的注解，请大家自行完成。

接下来，我们需要编写一个 Swagger 的 Spring 配置类，用于初始化 Springfox 所需的 Bean

实例。

```
package demo.msa.product.conf;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import springfox.documentation.builders.ApiInfoBuilder;
import springfox.documentation.builders.RequestHandlerSelectors;
import springfox.documentation.service.ApiInfo;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

@Configuration
@EnableSwagger2
public class SwaggerConf {

    @Value("${swagger.api.title}")
    private String apiTitle;

    @Value("${swagger.api.version}")
    private String apiVersion;

    @Value("${swagger.base-package}")
    private String basePackage;

    @Bean
    public Docket docket() {
        ApiInfo apiInfo = new ApiInfoBuilder()
            .title(apiTitle)
            .version(apiVersion)
            .build();
        return new Docket(DocumentationType.SWAGGER_2)
            .apiInfo(apiInfo)
            .select()
            .apis(RequestHandlerSelectors.basePackage(basePackage))
            .build();
    }
}
```


我们需要在 `SwaggerConf` 类上使用 Springfox 提供的 `@EnableSwagger2` 注解，表示使用 Swagger 2.0 规范来生成 REST API 文档。在 `SwaggerConf` 类中，我们需要创建一个 Springfox 所需的 `Docket` 对象，它用于封装 Swagger UI 所需的数据。`Docket` 对象需要传入一个 `ApiInfo` 对象，它表示 API 文档的基本信息，包括 API 标题、API 版本等信息，我们还需要告诉 Swagger 框架从哪个包路径下扫描 Swagger 注解，这些信息可通过 `@Value` 注解从 `application.properties` 配置文件中获取。

```
swagger.api.title=Product REST API
swagger.api.version=1.0.0
swagger.base-package=demo.msa.product.controller
```

启动 Spring Boot 应用，并在浏览器中访问 `http://localhost:8080/swagger-ui.html`，将看到以下 Swagger UI，即 REST API 文档，如图 6-22 所示。

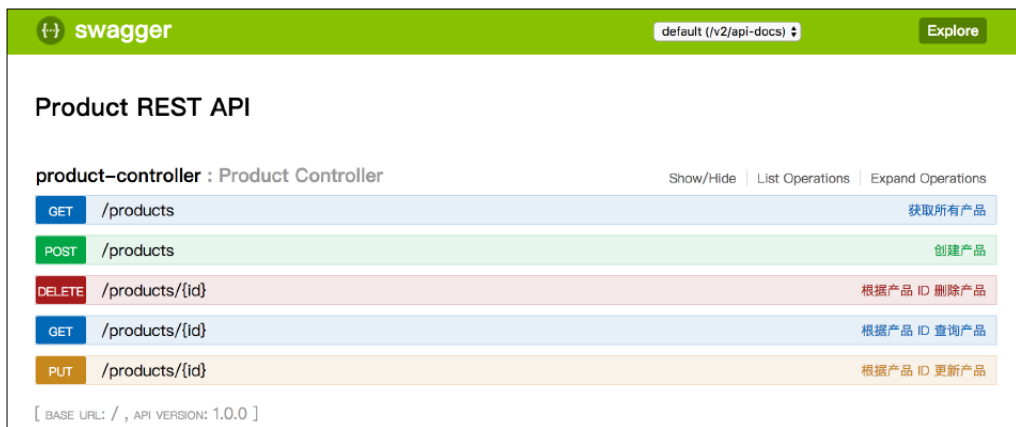


图 6-22 在 Spring Boot 应用中查看 Swagger UI

点击 `GET:/product/{id}` 这个 REST API，可查看它的详细信息，如图 6-23 所示。

此时，我们还能输入相应的请求参数，并点击“Try it out!”按钮来发送 REST 请求，此时将在下方输出相应的响应结果。

不仅如此，我们如果在浏览器中访问 `http://localhost:8080/v2/api-docs`，就能获取 Swagger 源文件，即 JSON 格式的 API 定义文档，如图 6-24 所示。

现在 REST API 文档已经生成了，我们需要与前端开发人员进行交流，确保每个 REST API 的输入和输出都是合理的。当这些沟通工作都完毕后，我们需要逐个填充尚未完成的 Controller 方法，并自行通过单元测试来确保 REST API 的可用性。

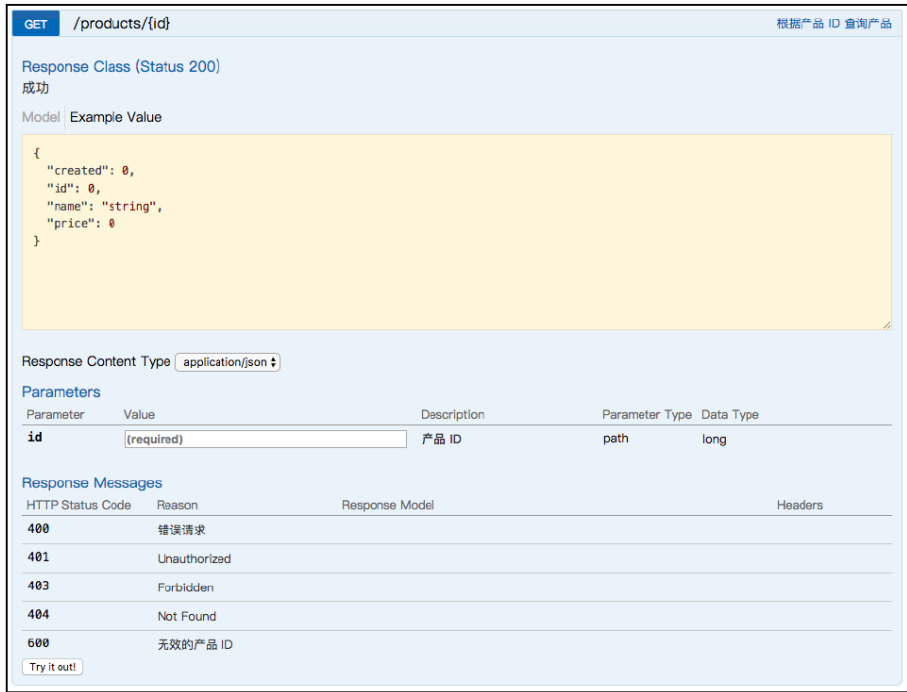


图 6-23 查看 REST API 详细信息



图 6-24 获取 Swagger 源文件

综上所述，我们要在 Spring Boot 应用中通过 Swagger 自动生成一份 REST API 文档，只需三个步骤。

- (1) 添加 Springfox 的 Maven 依赖。
- (2) 使用 Swagger 提供的 Java 注解，并在 Controller 方法上进行标注。
- (3) 通过 Swagger 提供的 Docket 来生成 Swagger UI。

虽然 Swagger 已经很优秀了，但我们不得不说的是，Swagger 并非业界最轻量级的 REST API 文档生成工具，因为它对已有应用存在一定的侵入性，例如，Maven 配置、Java 注解、Spring 配置等，这些对我们的代码都有一定的侵入。那么，是否存在更加轻量级的工具能够做到无侵入式地生成 REST API 文档呢？

此外，Springfox 只是将 REST API 文档“寄生”在现有的应用中，而无法做到生成的文档与运行的应用彼此分离，其实我们更希望的是文档可以基于代码生成出来，并且文档与代码是分离的，而且有一个静态站点能够聚合所有应用生成的文档，形成一个 REST API 文档站点。那么，是否存在更加合适的工具能够做到 REST API 文档与代码相分离呢？

我们发现了一款叫 apiDoc 的开源项目，它似乎比 Swagger 更加轻量级，同时还能自动生成静态文档站点。

6.3.2 REST API 文档的另一选择：apiDoc

apiDoc 基于 Node.js 开发，我们只需要在代码中使用注释（而不是注解），就能定义 REST API，并通过命令行工具来生成 REST API 文档。而且，apiDoc 还能支持几乎所有的编程语言。

通过 apiDoc 的官网，我们可快速了解它的使用方法，如图 6-25 所示。

apiDoc 官网：<http://apidocjs.com/>。

在使用 apiDoc 之前，我们要做的是使用 NPM 命令以全局的方式来安装 apiDoc 命令行工具。

```
npm install apidoc -g
```

安装完毕后，我们可通过 `apidoc -h` 命令来查看 apiDoc 命令行工具的使用方法。

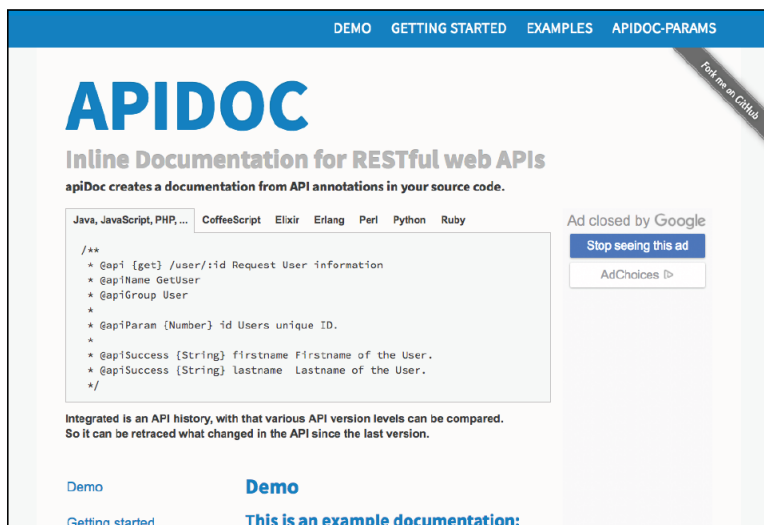


图 6-25 apiDoc 官网

接下来，我们只需在 Controller 方法上使用 apiDoc 提供的注释来定义 REST API 的基本信息，随后就能通过 apiDoc 命令来生成 REST API 文档，就像下面这样。

```
package demo.msa.product.controller;

...

@SuppressWarnings("JavaDoc")
@RestController
@RequestMapping(produces = "application/json")
public class ProductController {

    /**
     * @api {GET} /product/{id} 根据产品 ID 查询产品
     * @apiVersion 1.0.0
     * @apiGroup Product
     * @apiName getProductById
     *
     * @apiParam {Number} id 产品 ID
     *
     * @apiSuccess {Product} product 产品对象
     * @apiSuccess {String} product.id 产品 ID
     */
}
```

```

    * @apiSuccess {String} product.name 产品名称
    * @apiSuccess {Number} product.price 产品价格
    * @apiSuccess {Number} product.created 创建时间
    *
    * @apiError 400 错误请求
    * @apiError 600 无效的产品 ID
    *
    * @apiSuccessExample 输入
    * GET:/product/1
    * @apiSuccessExample 输出
    * {
    *   "id": 1,
    *   "name": "MacBook",
    *   "price": 10000,
    *   "created": xxx
    * }
    */
    @GetMapping("/product/{id}")
    public ResponseEntity<Product> getProductById(@PathVariable("id") long id) {
        // TODO: 尚未完成
        return ResponseEntity.badRequest().build();
    }

    ...
}

```

此时我们不再使用 Java 注解，而是使用 Java 注释的方式来定义 REST API，需要用到以下几种常用的注释指令。

- **@api**: 用于定义 REST API 的摘要信息。
- **@apiVersion**: 用于定义 REST API 的版本信息。
- **@apiGroup**: 用于定义 REST API 所属的分组。
- **@apiName**: 用于定义 REST API 对应的名称。
- **@apiParam**: 用于定义 REST API 的请求参数。
- **@apiSuccess**: 用于定义 REST API 成功返回时的数据对象，可定义为对象的层级结构。
- **@apiError**: 用于定义 REST API 的错误状态码与状态消息。

- `@apiSuccessExample`: 用于定义 REST API 成功调用的数据示例，可分别表示输入与输出。

由于以上注解指令并非 Java 规范所拥有，因此 IntelliJ IDEA 会出现黄色警告提示，若要去掉该警告，则需在类上添加`@SuppressWarnings("JavaDoc")`注解。

现在 REST API 定义完毕，下面就来生成 REST API 文档。

首先，我们在当前应用的根目录下创建一个名为 `apidoc.json` 的 JSON 文件，其内容如下。

```
{
  "name": "Product REST API",
  "version": "1.0.0"
}
```

然后，在源码根目录下执行以下 `apiDoc` 命令，将生成 REST API 文档。

```
apidoc -i src/main/java/demo/msa/product/controller -o ~/apidoc/web/product
```

其中，`-i` 选项用于设置解析 `apiDoc` 注释指令的源代码路径，一般定位源码中的 `Controller` 层目录；`-o` 指令用于设置输出 REST API 的静态站点目录，如图 6-26 所示。

```
$ ll ~/apidoc/web/product
total 200
-rw-r--r-- 1 huangyong staff 13091 6 16 11:06 api_data.js
-rw-r--r-- 1 huangyong staff 13071 6 16 11:06 api_data.json
-rw-r--r-- 1 huangyong staff 299 6 16 11:06 api_project.js
-rw-r--r-- 1 huangyong staff 290 6 16 11:06 api_project.json
drwxr-xr-x 3 huangyong staff 102 6 16 11:06 css
drwxr-xr-x 7 huangyong staff 238 6 16 11:06 fonts
drwxr-xr-x 3 huangyong staff 102 6 16 11:06 img
-rw-r--r-- 1 huangyong staff 27577 6 16 11:06 index.html
drwxr-xr-x 17 huangyong staff 578 6 16 11:06 locales
-rw-r--r-- 1 huangyong staff 28722 6 16 11:06 main.js
drwxr-xr-x 4 huangyong staff 136 6 16 11:06 utils
drwxr-xr-x 16 huangyong staff 544 6 16 11:06 vendor
```

图 6-26 apiDoc 静态目录

现在我们要做的是启动一个 Nginx 容器，将以上 `apiDoc` 静态站点目录挂载到该容器中。

```
docker run \
-d \
-p 80:80 \
-v ~/apidoc/web:/usr/share/nginx/html \
--name apidoc \
nginx
```

当容器启动后，我们在浏览器中访问 `http://localhost/product/`，就能查看 `apiDoc` 文档，如图 6-27 所示。

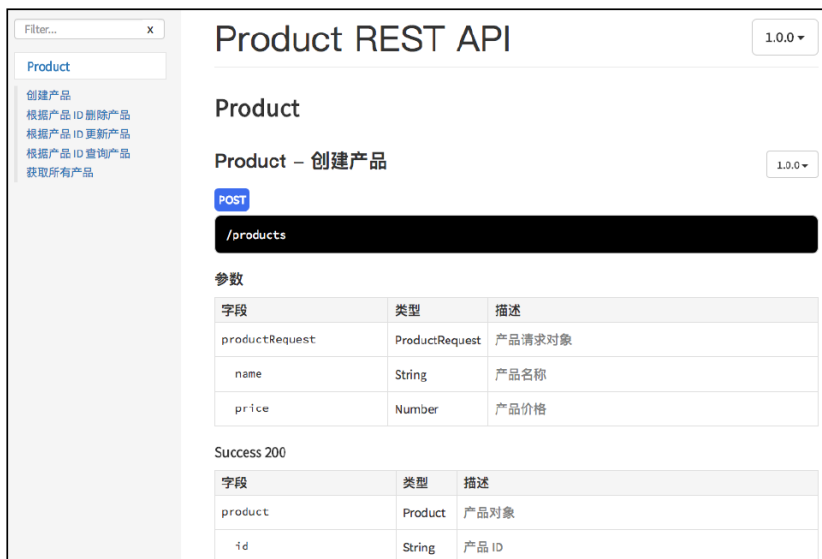


图 6-27 查看 apiDoc 文档

我们可将以上 `apiDoc` 命令放入 Jenkins 自动化构建过程中，这样就能做到在构建服务的同时，该服务对应的 REST API 文档也能自动生成，只需刷新一下浏览器，就能看到最新的 REST API 文档。

至此，一款轻量级的 REST API 文档站点基本搭建完毕，后续我们可针对此站点进行域名配置以及安全控制，将该站点地址分享给团队中所有的开发与测试人员，前期的沟通交流与后期的维护升级都能借助该文档，它正是我们微服务架构的 API 文档站点。

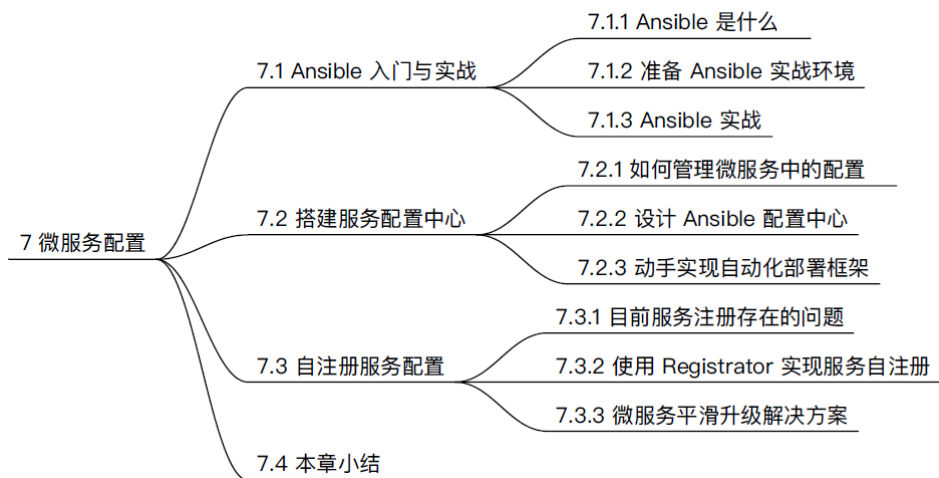
6.4 本章小结

本章的聚焦点集中在微服务 REST API 的测试方面，此外也对比了自动生成 REST API 文档的相关工具。首先我们以 Spring Boot 应用程序为例，分别针对 Service 层与 REST API 进行了单元测试，此时我们介绍了一种名为“测试驱动”的敏捷方法，该方法能帮助我们更加高效地实现代码逻辑。随后我们使用了 Postman 来充当 REST API 的测试工具，并结合 Jenkins 与 Newman 搭建了一款 REST API 的自动化测试框架，该框架可将测试用例与测试数据相分离，这也是我们所推崇的架构模式。最后我们分别使用了 Swagger 与 apiDoc 工具来自动生成 REST API 文档，并比较了这两款工具的优缺点，我们推荐大家使用 apiDoc，因为它更为轻量级，而且可基于源码快速生成 REST API 文档站点。

下一章我们将解决微服务应用程序中的配置问题，希望能将所有的配置参数都抽取出来并统一管理。

7 chapter

第 7 章 微服务配置



7.1 Ansible 入门与实战

在我们的微服务应用程序中，可能会存在一些配置信息（例如，MySQL、Redis、RabbitMQ 等连接配置），而且在不同的运行环境中，还可能会拥有多套不同的配置。如果我们将这些配置散落在应用程序内部，必然会增加一定的维护成本。可以想象，如果某个配置项发生了变化，我们此时必须手工更新大量的配置文件，这种方式既浪费时间又容易出错。我们更希望做到的是，将每个微服务中的配置进行统一管理，不仅能够给运维工作带来方便，而且还能确保敏感性配置不会泄漏，可见，统一配置可谓一举多得的事情。另一方面，我们目前可通过 Jenkins 从 Gitlab 中拉取源码，并使用 Maven 来编译源码并构建程序包与 Docker 镜像，随后将此 Docker 镜像上传到 Docker Registry 中，那么应该如何才能将此 Docker 镜像分发到指定的服务器上去运行 Docker 容器呢？

通过大量的技术调研，我们决定使用 Ansible，因为它既可用于集中化配置，也有利于自动化部署。

7.1.1 Ansible 是什么

Ansible 是一款开源的自动化运维工具，它基于 Python 语言开发，可用于配置系统与部署软件。Ansible 通过 SSH 方式并以“推模式”来操作远程服务器，我们无须在远程服务器中安装任何代理软件，该特性区别于传统的运维工具（例如，Chef 与 Puppet），同时做到了简单易用与安全可靠。

Ansible 一词出自一本科幻小说，它是一种虚构的以超光速传递信息的通信装置，可跨越任何距离同时控制无数飞船，就好像我们可控制海量远程服务器一样。

关于 Ansible 的更多介绍，请参见它官网，如图 7-1 所示。

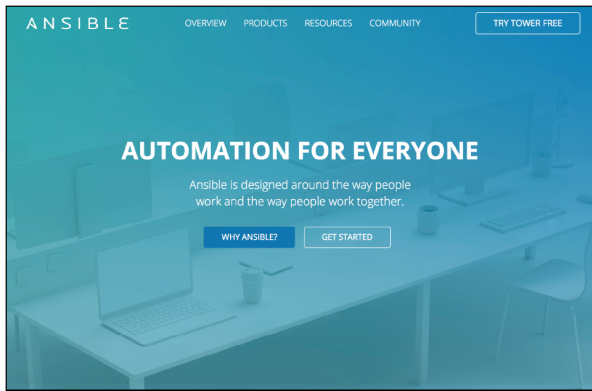


图 7-1 Ansible 官网

Ansible 官网：<https://www.ansible.com/>。

在官网首页可以看到，Ansible 公司的目标是为每个人实现自动化，这意味着 Ansible 的用户不仅包括运维人员，也包括开发和测试人员。

在开始使用 Ansible 之前，我们有必要对所需环境做好准备。

7.1.2 准备 Ansible 实战环境

我们需要准备两台服务器：Server1 与 Server2，并确保它们可以建立 SSH 连接，随后我们在 Server1 上执行 Ansible 命令，即可直接操作 Server2，如图 7-2 所示。



图 7-2 通过 SSH 连接两台服务器

假设 Server1 是我们的 Mac 主机，Server2 是一台 CentOS 服务器，它的 IP 地址是 192.168.199.215，我们需要在 Server1 中使用以下 ssh 命令以 root 用户身份来登录 Server2。

```
ssh root@192.168.199.215
```

随后系统将要求我们输入 root 用户的密码，从而进行身份认证。实际上，我们更希望每次进行 SSH 登录时无须输入任何密码，如何做到呢？

在 Server1 中，使用以下 ssh-keygen 命令来生成基于 RSA 加密算法的密钥库。

```
ssh-keygen -t rsa
```

密钥库生成完毕后，我们可在 Server1 的 ~/.ssh 目录下看到以下三个文件。

- id_rsa: 用于存放 SSH 私钥的文件。
- id_rsa.pub: 用于存放 SSH 公钥的文件。
- known_hosts: 用于存放曾经通过 SSH 登录的域名或 IP 地址。

我们只需将 Server1 的 ~/.ssh/id_rsa.pub 公钥文件中的文本内容复制到 Server2 的 ~/.ssh/authorized_keys 文件中，随后 Server2 将完全信任 Server1，从而实现 Server1 到 Server2 的 SSH 无密码登录。

在 Server1 中，使用以下 ssh-copy-id 命令快速完成公钥的复制操作。

```
ssh-copy-id root@server2
```

我们只需在 Server1 中安装 Ansible，如果 Server1 是 Mac 操作系统，那么可直接通过 Homebrew 来安装 Ansible。

```
brew install ansible
```

类似地，如果 Server1 是 CentOS Linux 操作系统，那么可通过 YUM 来安装 Ansible。

```
yum install ansible
```

安装完毕后，我们可立即查看 Ansible 版本号，从而确定是否安装成功。

```
ansible --version
```

```
ansible 2.3.1.0
  config file =
  configured module search path = Default w/o overrides
  python version = 2.7.13 (default, Apr 4 2017, 08:47:57) [GCC 4.2.1
Compatible Apple LLVM 8.1.0 (clang-802.0.38)]
```

接下来，我们将通过一系列实战过程，逐步掌握 Ansible 的使用方法。

7.1.3 Ansible 实战

7.1.3.1 验证服务器连通性

当 Server1 上的 Ansible 安装完毕后，我们可使用 Ansible 提供的 ping 模块来确定 Server1 与 Server2 的连通性，该步骤也是后续其他 Ansible 操作的基础。

Server1 向 Server2 发送“ping”信号，若 Server1 与 Server2 连接成功，则 Server1 将收到的“pong”信号，如图 7-3 所示。

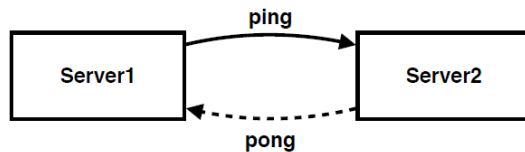


图 7-3 使用 Ansible 的 ping 模块

在使用 ping 模块之前，我们需要首先创建一个 Inventory（库存），它是 Ansible 的一个核心概念，它实际上是一份文件，用于存放所需连接的主机信息，内容可以包含域名或 IP 地址。

Ansible 提供了一份默认的 Inventory 文件，不同的操作系统，文件路径也不太一样。

- Mac: /usr/local/etc/ansible/hosts;
- Linux: /etc/ansible/hosts。

我们也可自行创建一份 Inventory 文件，不妨将其存放在当前目录下，并取名为 hosts，该文件的内容如下所示。

```
server2 ansible_host=192.168.199.215 ansible_user=root
```

Inventory 文件可存放多条主机信息，每条主机信息包括两部分。

(1) 主机别名: server2，对应 Server2 服务器。

(2) 主机变量: ansible_host 与 ansible_user，分别对应通过 SSH 登录 Server2 的 IP 地址与用户名。

接下来，我们使用以下 ansible 命令来连接 Server2 主机，并通过观察该命令所输出的信息来验证连接是否成功。

```
ansible server2 -i hosts -m ping
```

```
server2 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

以上信息表明，Server1 与 Server2 可成功建立连接。其中，-i 选项用于传入 Inventory 文件路径(当前目录下的 hosts 文件)，-m 选项用于设置需要执行的模块(Ansible 内置的 ping 模块)。

Inventory 文件还有一种更为流行的写法，也是我们所推荐的做法。

```
[servers]
192.168.199.215
```

我们创建了一个名为“severs”的主机群组，它可拥有多个主机地址，目前该群组中仅包括 Server2 的主机地址，此时 ansible_host 主机变量就可以省略，但 ansible_user 主机变量去哪儿了呢？

此时我们需要在运行 ansible 命令所在的当前目录下创建一个名为 ansible.cfg 的配置文件，其内容如下：

```
[defaults]
hostfile=hosts
remote_user=root
```

该文件中带有一个 **defaults** 配置组，其中包括两个变量。

(1) **hostfile**: 用于配置默认的主机文件路径。

(2) **remote_user**: 用于配置 SSH 登录用户名。

此时，我们只需输入以下 **ansible** 命令即可完成同样的 **ping-pong** 操作。

```
ansible servers -m ping
```

需要说明的是，我们一般会提供多个 **Inventory** 文件，分别对应不同的环境配置。例如，测试环境、预发环境、生产环境等。通过 **ansible** 命令强制传入的 **-i** 选项，将覆盖 **ansible.cfg** 配置文件中所定义的 **hostfile** 默认值。

7.1.3.2 使用 Ansible 安装 Nginx

如果在 **Server2** 上安装 **Nginx**，传统的做法是，首先在 **Server1** 上通过 **SSH** 登录 **Server2** 服务器，然后更新 **YUM** 软件包缓存，最后使用 **YUM** 安装 **Nginx** 软件包，大致包括以下命令。

```
ssh root@192.168.199.215
```

```
yum update
yum -y install nginx
```

需要注意的是，**Nginx** 不在 **CentOS** 基础软件源中，如果我们的 **CentOS** 是通过最小方式来安装的，那么需要自行添加 **EPEL** 软件源。推荐大家使用阿里云提供的 **EPEL** 软件源镜像，它能加快下载 **EPEL** 软件包的速度。只需执行以下命令就能获取阿里云的 **EPEL** 软件源镜像，使用时请注意对应的 **CentOS** 版本。

```
wget -O /etc/yum.repos.d/epel.repo http://mirrors.aliyun.com/repo/epel-7.repo
```

我们同样也能使用阿里云提供的 **CentOS** 软件源镜像，但操作之前最好备份一下默认的 **CentOS** 软件源镜像文件，最后别忘了重建 **YUM** 缓存，使最新的软件源镜像生效。

```
mv /etc/yum.repos.d/CentOS-Base.repo /etc/yum.repos.d/CentOS-Base.repo.backup
wget -O /etc/yum.repos.d/CentOS-Base.repo http://mirrors.aliyun.com/
repo/Centos-7.repo
yum makecache
```

然而，Ansible 只需在 Server1 中运行一条 `ansible` 命令，就能实现在远程服务器 Server2 中自动安装 Nginx，其结果与以上手工安装完全相同。

```
ansible servers -m yum -a "name=nginx update_cache=yes"
```

此时，我们通过 `-m` 选项来使用 `yum` 模块，该模块需要通过 `-a` 选项传入两个参数：`name`、`update_cache`。其中，`name` 参数用于表示需要安装的 YUM 软件包名称，`update_cache` 参数用于设置是否更新 YUM 缓存。

在 `ansible` 命令执行过程中是看不到任何日志信息的，除非在命令中添加 `-v`、`-vv`、`-vvv`、`-vvvv` 选项，我们可根据日志的详细程度自行选择。

当以上 `ansible` 命令执行成功后，我们可继续使用以下 `ansible` 命令来启动 Nginx 服务。

```
ansible servers -m service -a "name=nginx state=started"
```

此时，我们使用了 `service` 模块，它会自动在远程服务器上执行 `systemctl start nginx` 命令。

最后，我们还需要使用 `firewalld` 模块，将防火墙的 80 端口开启。

```
ansible servers -m firewalld -a "state=enabled port=80/tcp permanent=true immediate=yes"
```

以上 `ansible` 命令相当于在远程服务器上先后执行 `firewall-cmd --add-port=80/tcp --permanent` 与 `firewall-cmd --reload` 两条命令。

现在，我们打开浏览器中，在地址栏中输入 `http://192.168.199.215/`，即可看到 Nginx 默认首页。

Ansible 就是这样简单易用，只需执行几条命令，就能在远程服务器上完成我们所需的操作，包括但不限于安装软件。我们可在命令行中输入“`ansible-doc <模块名>`”命令查看指定模块的使用方法，也能在 Ansible 模块官方文档中查阅所有模块的使用方法。

Ansible 模块官方文档：http://docs.ansible.com/ansible/modules_by_category.html。

Ansible 除了以上操作命令，还提供了一种更为高效的使用方法。我们可将以上 `ansible` 命令当作不同的任务，并将这些任务编排到一份脚本文件中，随后可用于批量执行，Ansible 称这个批量脚本文件为 `Playbook`。下面我们就使用 `Playbook` 来改写以上 Nginx 的安装过程。

7.1.3.3 使用 Playbook 安装 Nginx

`Playbook` 是 Ansible 的脚本文件，它可基于 YAML 语法进行编写，通过“`ansible-playbook`

<YAML 文件>”命令来依次执行 YAML 文件中定义的相关任务。

我们之前在远程服务器上安装 Nginx 时，使用了以下三条 `ansible` 命令，随后我们会将它们编排到 Playbook 文件中。

```
ansible servers -m yum -a "name=nginx update_cache=yes"
ansible servers -m service -a "name=nginx state=started"
ansible servers -m firewalld -a "state=enabled port=80/tcp permanent=true
immediate=yes"
```

以下是整个远程安装 Nginx 的 Playbook 文件，它包括了以上所有 `ansible` 命令。

```
- name: Install Nginx
  hosts: servers
  tasks:
    - name: Installing Nginx Package
      yum:
        name: nginx
        update_cache: yes
    - name: Starting Nginx Server
      service:
        name: nginx
        state: started
    - name: Adding Nginx TCP Port
      firewalld:
        state: enabled
        port: "{{nginx_http_port}}/tcp"
        permanent: true
        immediate: yes
```

当前的 Playbook 文件中仅包含一个 Play(当然也能包含多个 Play)，该 Play 包括以下属性。

- (1) **name:** 用于描述该 Play 的名称。
- (2) **hosts:** 用于描述该 Play 需要连接的主机别名。
- (3) **tasks:** 用于管理需要执行的批量任务。

其中，每个任务又包含两种属性：

- (1) **name:** 用于描述当前的任务名称。
- (2) **模块:** 用于描述模块对应的名称及其相关参数。

需要注意的是，在“Adding Nginx TCP Port”任务中，我们使用了`{{nginx_http_port}}`占位符，它是一个主机变量，其具体配置在 `hosts` 文件中。

```
[servers]
192.168.199.215

[servers:vars]
nginx_http_port=80
```

我们在 `hosts` 文件中，通过`[servers:vars]`块来定义 `servers` 群组所包含的相关变量，我们称其为“群组变量”，这些变量可供主机组中所有的主机共享。一般情况下，我们会在 Playbook 文件中使用类似`{{nginx_http_port}}`的占位符，该占位符对应 `hosts` 文件中的配置参数，这样做是为了让配置与变量相分离，我们提倡这种分离思想。

此时，我们只需执行以下 `ansible-playbook` 命令就能在远程服务器中安装 Nginx。

```
ansible-playbook install-nginx.yml
```

随后，可看到以上命令输出的相关日志，包括了一个 `PLAY` 和四个 `TASK`，还有一个 `PLAY RECAP`。

```
PLAY [Install Nginx] *****

TASK [Gathering Facts] *****
ok: [192.168.199.215]

TASK [Installing Nginx Package] *****
changed: [192.168.199.215]

TASK [Starting Nginx Server] *****
changed: [192.168.199.215]

TASK [Adding Nginx TCP Port] *****
changed: [192.168.199.215]

PLAY RECAP *****
192.168.199.215      : ok=4    changed=3    unreachable=0    failed=0
```

其中，绿色的 `ok` 表示任务执行成功并对服务器没有任何影响，黄色的 `changed` 表示任务执行成功但对服务器做了一些变化，`unreachable` 表示未能连接成功，`failed` 表示操作失败。

同样地，我们也能编写一个卸载 Nginx 的 Playbook。

```
- name: Uninstall Nginx
  hosts: servers
  tasks:
    - name: Stoping Nginx Server
      service:
        name: nginx
        state: stopped
    - name: Removing Nginx Package
      yum:
        name: nginx
        state: absent
    - name: Removing Nginx TCP Port
      firewallld:
        state: disabled
        port: "{{nginx_http_port}}/tcp"
        permanent: true
        immediate: yes
```

执行以下 `ansible-playbook` 命令，可在远程服务器上卸载 Nginx。

```
ansible-playbook uninstall-nginx.yml
```

那么，Docker 是否也能远程安装呢？我们继续下去。

7.1.3.4 远程安装 Docker

我们编写一个安装 Docker 的 Playbook 文件，它包括以下五个任务：

- (1) 删除已安装的 Docker 软件包（为了安装最新的 Docker）。
- (2) 添加最新版本的 Docker 软件源。
- (3) 安装 Docker 软件包。
- (4) 启动 Docker 服务。
- (5) 将 Docker 服务加入开机启动项。

以下是整个远程安装 Docker 的 Playbook 文件，它用到了 `yum`、`get_url`、`service` 等 Ansible 模块。

```
- name: Install Docker
```

```
hosts: servers
tasks:
  - name: Uninstalling Old Packages
    yum:
      name: "{{item}}"
      state: absent
    with_items:
      - docker
      - docker-common
      - docker-engine
      - docker-selinux
      - docker-engine-selinux
      - container-selinux
  - name: Adding Docker Repo
    get_url:
      url: https://download.docker.com/linux/centos/docker-ce.repo
      dest: /etc/yum.repos.d/docker-ce.repo
      force: yes
  - name: Installing Docker Package
    yum:
      name: docker-ce
      state: present
      update_cache: yes
  - name: Starting Docker Server
    service:
      name: docker
      state: started
  - name: Adding Auto Start
    command: systemctl enable docker
```

需要注意的是，在“Uninstalling Old Packages”任务中，我们使用了“{{item}}”占位符，它的具体内容将从 `with_items` 列表中获取，此时可循环执行该任务，依次删除所提供的软件包。

执行以下 `docker-playbook` 命令，可在远程服务器上安装 Docker。

```
ansible-playbook install-docker.yml
```

随后我们可自行登录远程服务器，验证 Docker 服务是否安装成功。

为了提高 Docker 镜像的下载速度，我们可使用阿里云提供的 Docker 镜像加速器，具体使

用方法请参见阿里云 Docker Hub 镜像站点。

阿里云 Docker Hub 镜像站点: <https://cr.console.aliyun.com/>。

当我们拿到自己专属的加速器地址后,可在当前目录下创建一个名为 `daemon.json` 的文件,该文件内容如下。

```
{
  "registry-mirrors": [
    "https://xxx.mirror.aliyuncs.com"
  ]
}
```

以上 Docker 镜像加速器地址仅为示例,请大家使用自己的专属地址。

实际上, `daemon.json` 文件是 Docker 引擎 (Docker Daemon) 可以识别的配置文件,我们只需将其放入 `/etc/docker` 路径下,并重启 Docker 服务即可生效。

我们现在就来编写一个 Playbook,为现在已运行的 Docker 服务配置镜像加速。

```
- name: Config Docker
  hosts: servers
  tasks:
    - name: Copying Docker Daemon File
      copy:
        src: daemon.json
        dest: /etc/docker/daemon.json
        mode: 0600
    - name: Restrating Docker Server
      service:
        name: docker
        state: restarted
```

在“Copying Docker Daemon”任务中,我们使用了 `copy` 模块,将 Playbook 所在目录下的 `daemon.json` 文件复制到远程服务器中,文件绝对路径为 `/etc/docker/daemon.json`。

执行以下 `ansible-playbook` 命令,可在远程服务器上配置 Docker 加速器。

```
ansible-playbook config-docker.yml
```

接下来,我们尝试在远程服务器上安装 Nginx 的 Docker 容器,以实现容器化自动部署。

7.1.3.5 远程运行 Nginx 容器

我们可使用 Ansible 提供的 `docker_container` 模块来启动 Nginx 的 Docker 容器。

```
- name: Start Nginx Container
  hosts: servers
  tasks:
    - name: Starting Nginx Container
      docker_container:
        name: nginx
        image: nginx
        ports:
          - "{{nginx_http_port}}:80"
```

执行以下 `ansible-playbook` 命令，本想在远程服务器上运行 Nginx 的 Docker 容器，但此时却报错了。

```
ansible-playbook start-nginx-container.yml

...
fatal: [192.168.199.215]: FAILED! => {"changed": false, "failed": true,
"msg": "Failed to import docker-py - No module named requests.exceptions. Try
`pip install docker-py`"}
...
```

从报错现象来看，应该是远程服务器上未安装 `docker-py` 软件包，它是一个基于 Python 的 Docker 客户端，我们同样也可通过 Playbook 的方式来自动安装。

```
- name: Install Docker SDK for Python
  hosts: servers
  tasks:
    - name: Installing docker-py Package
      pip:
        name: docker-py
```

执行以下 `ansible-playbook` 命令，本想在远程服务器上安装 `docker-py`，没想到此时又报出了新的错。

```
ansible-playbook install-docker-py.yml
```

```
...
fatal: [192.168.199.215]: FAILED! => {"changed": false, "failed": true,
"msg": "Unable to find any of pip2, pip to use.  pip needs to be installed."}
...
```

此报错说明远程服务器上并未安装 `pip` 命令，因此无法通过 Ansible 所提供的 `pip` 模块来安装 `docker-py`。为了尽量少地在服务器上安装过多的软件包，我们可继续使用 `yum` 模块来安装 `docker-py`，只是在 YUM 中，该软件包名为 `docker-python`。

```
- name: Install Docker SDK for Python
  hosts: servers
  tasks:
    - name: Installing docker-python Package
      yum:
        name: docker-python
        update_cache: yes
```

执行以下 `ansible-playbook` 命令，可在远程服务器上安装 `docker-python`。

```
ansible-playbook install-docker-py.yml
```

当 `docker-python` 软件包安装完毕后，我们再次使用 Playbook 执行 `start-nginx-container.yml` 脚本，将成功地在远程服务器上运行 Nginx 容器，可通过浏览器访问 `http://192.168.199.215/`，查看 Nginx 的默认首页。

如果执行 Ansible 命令失败后，不仅可看到相应的错误提示，Ansible 还会生成一份后缀为 `retry` 的文件，我们称其为“重试文件”。当我们将 Playbook 调整完毕后，可在命令后添加 `--limit` 选项来指定重试文件，此时将从上次失败的任务开始执行。如果我们不太希望 Ansible 为我们自动生成的重试文件，可在 `ansible.cfg` 配置文件中的 `[defaults]` 块中添加 `retry_files_enabled=False` 配置，将该功能禁用。

最后需要补充说明的是，如果我们反复执行同样的 Playbook，此时 Ansible 会确保重复执行的操作不会影响现有环境。例如，当我们反复执行 `install-docker.yml` 时，此时不会在远程服务器上重复安装 Docker 服务。也就是说，Playbook 可以反复执行，其结果都是一样的，我们称这样的特性为“幂等性”，Ansible 可以更加聪明完成我们交给它的任务。

通过启动 Nginx 容器这个案例中我们不难发现，在执行 `start-nginx-container.yml` 之前，我们需要首先执行 `install-docker.yml` 与 `install-docker-py.yml`，而且必须按照这个顺序才能成功启动 Nginx 容器。那么，是否有一种方式，能够将 Playbook 的执行顺序编排在一起呢？Ansible 官方

推荐我们使用 Role 结构来使用 Playbook，它不仅可以编排 Playbook 的执行顺序，还能让我们的 Playbook 更加有条理性，更容易维护和扩展。

7.1.3.6 使用 Role 结构改造 Playbook

以 start-nginx-container.yml 为例，我们下面将对其改造为 Role 结构的 Playbook，以前的 Playbook 配置如下所示。

```
- name: Start Nginx Container
  hosts: servers
  tasks:
    - name: Starting Nginx Container
      docker_container:
        name: nginx
        image: nginx
        ports:
          - "{{nginx_http_port}}:80"
```

我们可将 tasks 片段抽取出来，将其放入 roles/start-nginx-container/tasks/main.yml 文件中。其中，start-nginx-container 就是一个具体的 role 名称。

```
- name: Starting Nginx Container
  docker_container:
    name: nginx
    image: nginx
    ports:
      - "{{nginx_http_port}}:80"
```

这里的{{nginx_http_port}}占位符可从 roles/start-nginx-container/defaults/main.yml 文件中获取，该文件用于存放当前 role 作用域下的所有默认变量。

```
nginx_http_port: 80
```

同时，我们可将以前 hosts 文件中的[servers:vars]块删除。

```
[servers]
192.168.199.215

{servers:vars}
nginx_http_port=80
```

现在的 start-nginx-container.yml 文件将变得更加精简，我们将以前的 tasks 改为 roles，此时只需在 roles 中添加 start-nginx-container 所依赖的 role 即可。

```
- name: Start Nginx Container
  hosts: servers
  roles:
    - install-docker
    - install-docker-py
    - start-nginx-container
```

Playbook 目录发生了明显的变化，以前的目录结构更加扁平，现在的目录结构更加规范，如图 7-4 所示。

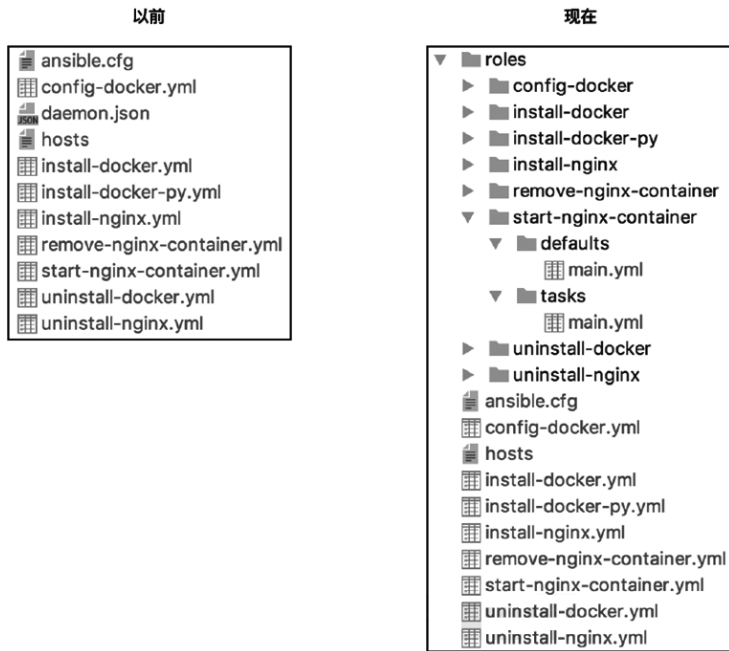


图 7-4 Playbook 目录变化对比

此时我们可在一台全新的服务器上直接执行 start-nginx-container.yml，首先会执行 install-docker 与 install-docker-py 包含的任务，最后才会执行 start-nginx-container 自身包含的任务，我们轻松地通过 Role 结构将有依赖关系的 Playbook 进行了编排。

关于 Ansible 的高级用法还有很多，这里我们仅对它的基本用法进行了讨论。如果大家想深度学习 Ansible，不妨阅读它的官方文档。

Ansible 官方文档: <https://docs.ansible.com/ansible/index.html>。

在下一节中，我们将充分发挥 Ansible 的价值，将其作为微服务的“配置中心”，它与 Jenkins、Gitlab、Maven、Docker Registry 等工具进行整合，可将微服务自动化部署发挥到极致。

7.2 搭建服务配置中心

通过上一节的学习，我们已经对 Ansible 有了一定的了解，知道如何使用 Playbook 批量执行 Ansible 任务，从而完成我们希望执行的操作。在本节中，我们将使用 Ansible 来搭建一款轻量级的微服务“配置中心”，并整合 Jenkins、Gitlab、Maven、Docker Registry 等工具，让微服务的部署过程变得更加自动化。整个部署过程仍然由 Jenkins 来主导，Jenkins 首先从 Gitlab 中拉取项目源码，随后会调用 Maven 来完成编译与打包，接着调用 Ansible 配置中心来获取指定运行环境下的服务配置，还会通过 Ansible 生成 Docker 镜像，最后在远程服务器上启动相应的 Docker 容器。整个部署过程无须人工参与，可做到完全自动化。

下面，我们将实现这款全自动化的微服务部署流程，大家可以清楚地看到 Ansible 在其中所扮演的重要角色。

7.2.1 如何管理微服务中的配置

我们已经用 Spring Boot 框架实现了一个微服务，该服务需要连接 MySQL 数据库，因此我们需要在 application.properties 配置文件中添加一些关于 MySQL 的数据源配置。

```
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/demo?useSSL=false
spring.datasource.username=root
spring.datasource.password=root
```

以上配置仅限于连接本地 MySQL 数据库，适合在本地做开发或测试，但如果需要将其部署到生产环境中，那么以上这些配置参数显然需要进行调整，手工替换配置参数肯定不是明智之举。

我们最容易想到的解决方案是利用 Maven Profile 技术来解决，此时需要在 application.properties 配置文件中使用 Spring Boot 提供的@...@占位符，将其替换掉所有的配置参数。

```
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://@mysql_address@/@database_name@?useSS
```



```
L=false
spring.datasource.username=@mysql_username@
spring.datasource.password=@mysql_password@
```

随后需要在 `pom.xml` 配置文件中，针对不同的运行环境，分别定义相应的配置参数，可通过 `project.profiles.profile.id` 节点来指定所对应的环境名称。比如，开发环境对应的 Profile ID 是 `development`，生产环境对应的 Profile ID 是 `production`，实际情况可能还有更多的运行环境。

```
<project>
...
<profiles>
  <profile>
    <id>development</id>
    <activation>
      <activeByDefault>true</activeByDefault>
    </activation>
    <properties>
      <mysql_address>localhost:3306</mysql_address>
      <mysql_username>root</mysql_username>
      <mysql_password>root</mysql_password>
      <database_name>demo</database_name>
    </properties>
  </profile>
  <profile>
    <id>production</id>
    <properties>
      <mysql_address>...</mysql_address>
      <mysql_username>...</mysql_username>
      <mysql_password>...</mysql_password>
      <database_name>...</database_name>
    </properties>
  </profile>
</profiles>
...
</project>
```

当 Jenkins 自动构建时，会调用 Maven 的 `mvn package` 命令，并针对指定的 Profile 来构建相应的 jar 包。例如，构建生产环境下的 jar 包，只需运行 `mvn package -Pproduction` 命令即可，

如果不带-P 选项，默认将构建适用于开发环境（development）的 jar 包。

以上方案虽然基本上能解决我们遇到的多环境配置问题，但这些配置项仍然存放在项目源码中，尤其是生产环境的配置，这种方式明显很不安全。

我们更希望做到的是，将参数配置从项目源码中抽取出来，并统一将其放入一个叫“配置中心”的地方。当我们需要对该项目进行构建时，Jenkins 将从配置中心中获取指定环境下的相关配置参数，并生成适用于该环境的 jar 包，随后将其加入 Docker 镜像，并上传到 Docker Registry 中，以便后续可在指定环境下启动 Docker 容器。

Ansible 将扮演这个配置中心的角色，它将于 Jenkins 强强联手，协作完成微服务的自动化部署工作。

7.2.2 设计 Ansible 配置中心

我们首先通过一张图来说明 Jenkins 与 Ansible 相集成，实现微服务自动化部署的整个流程，如图 7-5 所示。

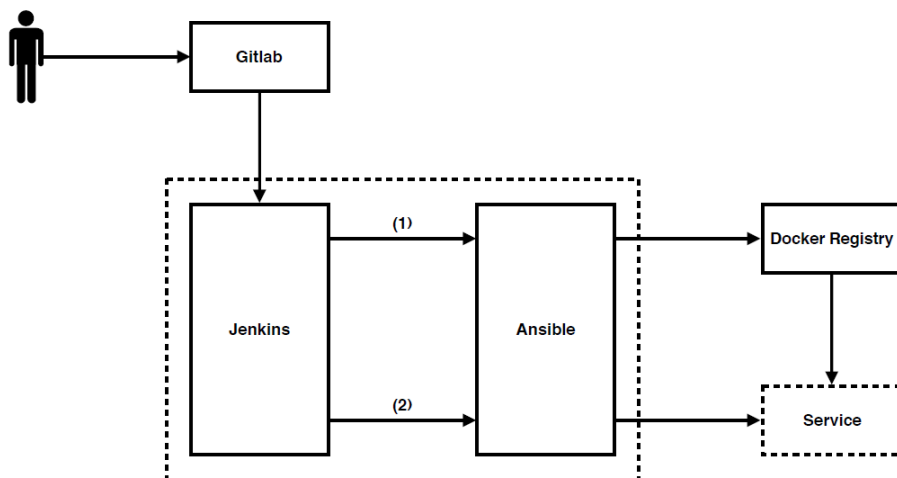


图 7-5 集成 Jenkins 与 Ansible 实现微服务自动化部署

Jenkins 与 Ansible 必须部署在同一台服务器上，这样更加有利于 Jenkins 直接调用 Ansible，但 Gitlab 与 Docker Registry 可以分布在其他服务器上，通过 Ansible 生成的 Service 容器也能在单独的服务器上运行。

当开发人员将代码推送至 Gitlab 后，Jenkins 将自动检测出 Gitlab 已有的最新代码，此时可将其拉取到 Jenkins 所在的服务器上，随后调用 Maven，编译项目源码并将其构建为 jar 包。接下来，Jenkins 将通过以下两个步骤来完成微服务的自动化部署。

(1) 调用 Ansible, 在本地生成指定运行环境的配置文件, 随后立即生成 Docker 镜像并将其上传到 Docker Registry 中。

(2) 再次调用 Ansible, 在远程启动 Docker 容器, 在启动容器之前会首先从 Docker Registry 中拉取对应的 Docker 镜像。

第一个步骤将使用 Ansible 生成两份配置文件, 第一份是 Spring Boot 应用程序在指定运行环境下的 `application.properties`, 第二份是生成 Docker 镜像所需的 `Dockerfile`。我们可在 `Dockerfile` 文件中将 Maven 构建的 jar 包与 Ansible 生成的 `application.properties` 文件一起复制到 Docker 镜像中, 并在启动 Spring Boot 应用程序的 `java -jar` 命令中添加 `--spring.config.location` 选项, 用于覆盖 jar 包内默认的 `application.properties`。随后, 将通过 Ansible 所提供的 `docker_image` 模块来构建 Docker 镜像, 并将其推送至 Docker Registry 中进行存储。

第二个步骤更加简单, 我们只需通过 Ansible 来连接指定环境的远程服务器, 并使用 Ansible 所提供的 `docker_container` 模块, 从 Docker Registry 中拉取对应的 Docker 镜像并启动 Docker 容器。我们要确保每次都会启动一个新的容器, 同时移出已有的同名容器。也就是说, 对于某个服务而言, 镜像可以多个, 但正在运行中的容器只能有一个。

下面, 我们就一起动手实现这个自动化部署框架, 体验 Jenkins 与 Ansible 相集成的强大威力。

7.2.3 动手实现自动化部署框架

首先, 我们在 Jenkins 所在的服务器上创建一个名为 `ansible-playbook` 的空目录, 用于存放所有 Ansible Playbook 的配置文件。不妨将该目录放到 `root` 用户的根目录下 (可通过 `~/ansible-playbook` 路径来访问), 这样有利于 Jenkins 直接调用。可使用 Ansible Role 规范创建以下 Playbook 目录结构, 如图 7-6 所示。

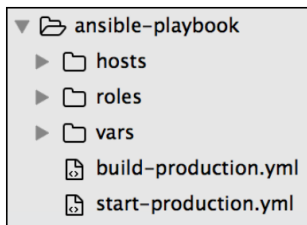


图 7-6 ansible-playbook 目录结构

在 `ansible-playbook` 根目录下有两个 Playbook 脚本, Jenkins 随后可直接执行它们。

(1) `build-production.yml`: 用于构建一个运行在生产环境下的 Docker 镜像。

(2) `start-production.yml`: 用于启动一个运行在生产环境下的 Docker 容器。

除了以上两个 Playbook 脚本，还有三个目录。其中，`roles` 目录是 Ansible Role 规范中所提供的标准目录，而 `hosts` 与 `vars` 却是我们自定义的目录。

(1) `roles`: 用于存放 Playbook 需要执行的具体任务及其相关文件。

(2) `hosts`: 用于存放对应运行环境的主机信息，包括 IP 地址、SSH 用户名等。

(3) `vars`: 用于存放 `roles` 目录下的配置文件所需的相关变量（配置参数）。

我们首先完成 `hosts/production.yml` 文件，它用于存放生产环境服务器所需的相关配置。

```
production ansible_host=192.168.199.219 ansible_user=root
```

我们定义了一个别名为 `production` 的主机，其中的 `ansible_host` 与 `ansible_user` 是主机参数，它们分别对应登录 SSH 所需的 IP 地址与用户名。这里定义的一个别名 `production`，可在 Playbook 的 `hosts` 中使用，用于连接指定的服务器。Ansible 也提供了一个默认的别名——`localhost`，我们可用这个别名来连接 Ansible 所在的本地服务器。

在 `vars/production.yml` 文件中存放了用于生产环境的相关配置，所有的配置参数采用 `key-value` 结构来定义，我们可在 Playbook 的相关任务中通过 `{{key}}` 来获取对应的 `value`。

```
docker_registry_address: 192.168.199.215:5000

mysql_address: 192.168.199.215:3306
mysql_username: root
mysql_password: root
database_name: demo
```

当 `hosts` 与 `vars` 目录及其文件都准备就绪后，我们来编写需要给 Jenkins 调用的 `build-production.yml` 脚本。

```
- name: Copy Config and Bulid Image
  hosts: localhost
  roles:
    - build-image
  vars_files:
    - vars/production.yml
```

我们使用 `build-production.yml` 脚本在本地（`localhost`）复制配置文件并构建 Docker 镜像，其中 `build-image` 角色的任务脚本在 `roles/build-image/tasks/main.yml` 文件中，它需要从

vars/production.yml 文件中获取相应的配置参数。

下面我们来创建 build-image 角色所需的目录结构，如图 7-7 所示。

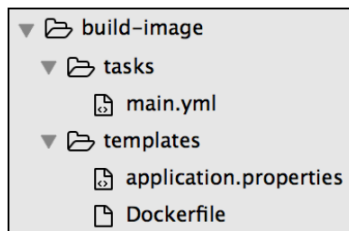


图 7-7 build-image 角色目录结构

build-image 角色需要完成的目标是将 templates 目录下的两份配置文件，分别与 vars/production.yml 参数文件合并，并将其最终生成的配置文件复制到 Docker 镜像中。

在 application.properties 模板中包含了一些配置参数的占位符，它们的具体参数值可从 vars/production.yml 参数文件获取。

```

spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://{{mysql_address}}/{{database_name}}?useSSL=false
spring.datasource.username={{mysql_username}}
spring.datasource.password={{mysql_password}}

```

在 Dockerfile 文件中需要将运行 Spring Boot 应用程序的 jar 包与以上 application.properties 文件分别复制到 Docker 镜像中。

```

FROM openjdk:alpine
WORKDIR /root
COPY {{JOB_NAME}}.jar app.jar
COPY application.properties app.properties
EXPOSE 8080
CMD java -jar app.jar --spring.config.location=app.properties

```

首先我们使用 openjdk:alpine 作为基础镜像，并将 /root 目录作为镜像内部的工作目录，后续所有的 Dockerfile 指令（例如，COPY 与 CMD）将基于该目录进行操作。其中，{{JOB_NAME}} 参数将从 Jenkins 中传入，表示 Jenkins 的 Job 名称，我们希望它也是所构建的服务名称。随后我们将 Dockerfile 所在目录下的 {{JOB_NAME}}.jar 与 application.properties 分别复制到将要构建的 Docker 镜像中，并它们进行重命名。接着我们声明了一个允许暴露到容器外部的端口号

8080,在启动容器时需要进行宿主机与容器的端口映射。最后我们使用 CMD 指令来执行 `java -jar` 命令,此时需要传入 `--spring.config.location=app.properties` 选项,目的是让 `app.properties` 文件取代 `app.jar` 中默认提供的 `application.properties` 文件。

强烈推荐使用 `openjdk:alpine` 基础镜像,而不要使用 `openjdk:7`,因为前者的镜像体积比后者要小得多,同时也能够满足我们的需求。经过验证,基于 `openjdk:7` 所构建的 Docker 镜像需要 736MB,而基于 `openjdk:alpine` 所构建的 Docker 镜像只有 117MB,使用 `openjdk:alpine` 明显可节省了大量的磁盘空间。

同时也强烈建议对变量命名做一个规范,大写的变量名为 `Jenkins` 变量,小写的变量名为 `Ansible` 变量,这样更加容易识别和维护。

当以上模板文件都准备完毕后,我们来编写 `roles/build-image/tasks/main.yml` 任务文件,它包括以下三个任务。

```
- name: Merging and Copying Dockerfile
  template:
    src: Dockerfile
    dest: "{{TARGET_PATH}}/Dockerfile"

- name: Merging and Copying application.properties
  template:
    src: application.properties
    dest: "{{TARGET_PATH}}/application.properties"

- name: Building and Pushing Docker Image
  docker_image:
    name: "{{docker_registry_address}}/{{ENV}}/{{JOB_NAME}}:{{BUILD_NUMBER}}"
    path: "{{TARGET_PATH}}"
    push: yes
    force: yes
```

前两个任务用于将 `templates` 目录下的 `Dockerfile` 与 `application.properties` 文件分别复制到 `{{TARGET_PATH}}` 目录下,该参数也由 `Jenkins` 传入。第三个任务用于构建 Docker 镜像,并将其推送至 Docker Registry 中。其中, `{{ENV}}` 与 `{{BUILD_NUMBER}}` 参数都由 `Jenkins` 所传入,前者是该镜像对应的环境名,后者是一个自增长的构建编号。需要注意的是,我们将所生成的 Docker 镜像名称中不仅包括 Docker Registry 的地址作为前缀,还需包括具体的环境名称,以便识别出该镜像可适用的环境,最终构建的 Docker 镜像名称与 “`192.168.199.215:5000/production/product-service:1`” 类似。可见,这里所有大写命名的变量都由 `Jenkins` 传入,它与 `Ansible`

以小写命名的变量有明显的区分度。

下面，我们来创建 `start-container` 角色所需的目录结构，如图 7-8 所示。

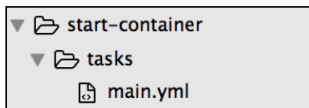


图 7-8 `start-container` 角色目录结构

当完成了 `build-production.yml` 脚本后，我们来编写 `start-production.yml` 脚本，它用于在 `production` 服务器上启动 `Docker` 容器。

```

- name: Start Docker Container
  hosts: production
  roles:
    - start-container
  vars_files:
    - vars/production.yml
  
```

接着我们来编写 `roles/start-container/tasks/main.yml` 任务文件，它仅包括一个任务，即启动 `Docker` 容器。

```

- name: Starting Docker Container
  docker_container:
    name: "{{JOB_NAME}}"
    image: "{{docker_registry_address}}/{{ENV}}/{{JOB_NAME}}:{{BUILD_NUMBER}}"
    ports:
      - "{{SERVICE_PORT}}:8080"
  
```

由于每次 `Jenkins` 构建时都会产生新的 `BUILD_NUMBER`，那么所构建的镜像名称也会发生变化，因此每次都能启动一个全新的容器。换句话说，在新容器启动前，旧容器将自动被移出，不会在同一个 `Docker` 引擎中同时运行多个相同的容器。

在 `Ansible Playbook` 的任务脚本中，我们使用了 `Jenkins` 变量。例如，`JOB_NAME`、`BUILD_NUMBER`、`ENV`、`TARGET_PATH`、`SERVICE_PORT` 等，这些变量是怎样传递进来的呢？

我们现在进入 `Jenkins`，首先创建一个名为“`product-service`”的 `Job`，随后可在添加两个构建参数，如图 7-9 所示。

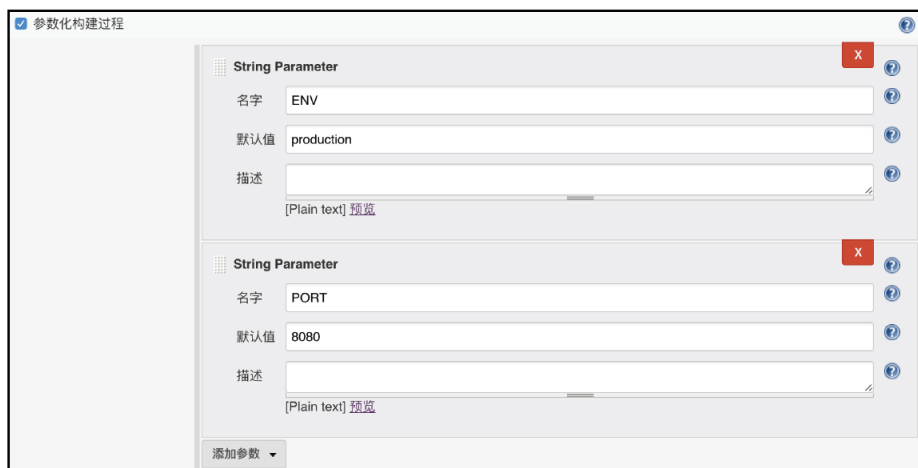


图 7-9 添加 Jenkins 构建参数

ENV 参数表示当前构建过程所对应的环境名, PORT 参数表示所启动 Docker 容器对外暴露的 HTTP 端口号。

接着我们添加 Git 仓库的地址与登录信息, 并配置一个 Poll SCM 表达式, 用于定时从 Git 仓库中拉取源码。

最后我们添加一段 Excute shell 脚本, 该脚本将被 Jenkins 自动执行。

```
mvn clean package

ansible-playbook \
~/ansible-playbook/build-$ENV.yml \
-i ~/ansible-playbook/hosts/$ENV.yml \
-e "JOB_NAME=$JOB_NAME BUILD_NUMBER=$BUILD_NUMBER ENV=$ENV TARGET_PATH=
$(PWD)/target"

ansible-playbook \
~/ansible-playbook/start-$ENV.yml \
-i ~/ansible-playbook/hosts/$ENV.yml \
-e "JOB_NAME=$JOB_NAME BUILD_NUMBER=$BUILD_NUMBER ENV=$ENV SERVICE_PORT=
$PORT"
```

以上脚本分为三部分, 第一部分用于调用 Maven 来编译源码并打成 jar 包, 第二部分用于调用 Ansible Playbook 脚本, 此时需要传入一个 Inventory 文件路径 (通过 -i 选项), 更重要的是传入了一个额外的配置参数列表 (通过 -e 选项), 该参数列表包括多个参数, 每个参数通过

key=value 结构来表示，多个参数用空格分隔。通过-e 选项传入的 Jenkins 参数可在 Ansible Playbook 的任务脚本中直接使用，它也是 Jenkins 与 Ansible 传递参数的常用方式。

保存以上 Jenkins Job 配置，现在是时候跑一遍 Jenkins Job 了。

点击“Build with Parameters”链接，此时 Jenkins 会提示我们确定默认的构建参数，如图 7-10 所示。



The image shows a Jenkins web interface for a project named 'Project product-service'. It prompts the user to provide parameters for building the project. There are two input fields: 'ENV' with the value 'production' and 'PORT' with the value '8080'. Below these fields is a blue button labeled '开始构建' (Start Build).

图 7-10 确认 Jenkins 构建参数

点击“开始构建”按钮，将进入 Jenkins 构建过程，我们可在“Console Output”中观察构建过程中输出的日志信息。

情况不妙，我们遇到了一个错误，Jenkins 构建失败了：

```
fatal: [localhost]: FAILED! => {"changed": false, "failed": true, "msg":
"Error pushing image 192.168.199.215:5000/production/product-service: Get
https://192.168.199.215:5000/v1/_ping: http: server gave HTTP response to HTTPS
client"}
```

该错误表明本地无法访问远程的 Docker Registry，由于 Docker Registry 为了确保访问安全性，要求我们通过 HTTPS 访问，当然这只是默认行为，我们也能禁用该安全功能，而直接使用 HTTP 来访问。如果在局域网内部访问 Docker Registry，我们可以忽略该安全限制。只需打开 /etc/docker/daemon.json 文件，在之前已添加的 registry-mirrors 配置后面添加一段 insecure-registries 配置即可，此时表示忽略 Docker Registry 的 HTTPS 安全访问。

```
{
  "registry-mirrors": [
    "https://xxx.mirror.aliyuncs.com"
  ],
  "insecure-registries": [
    "192.168.199.215:5000"
  ]
}
```

```
    ]  
}
```

需要注意的是，我们一定要重启 Docker 服务，才能使以上配置生效。

再次手工执行一次 Jenkins 构建，随后将看到最终构建成功的信息。此时 Docker 镜像已创建完毕，Docker 容器也在生产环境中部署成功。

现在我们只需修改项目源码，并将源码提交并推送至 Gitlab 中，随后 Jenkins 将自动构建，最新修改的代码将自动拉取、编译、打包、生成配置、构建镜像、启动容器，整个部署过程可做到完全自动化。

7.3 自注册服务配置

微服务的配置实际上包括两部分，一部分是微服务内部应用程序的配置（例如，数据库连接方式等配置参数），这些内部配置可使用 Ansible 进行管理；另一部分是我们之前谈到的微服务外部容器相关的配置（包括服务名称、IP 地址、端口号等），这些外部配置可使用 ZooKeeper 进行管理。对于内部配置而言，我们可做到配置与应用相分离，也就是说，配置参数完全集中在 Ansible 配置中心。然而，对于外部配置而言，我们目前却需要将应用依赖于 ZooKeeper 客户端，这种解决方案明显存在问题。

现在我们就来解决应用外部配置问题，同样做到应用与配置相分离。

7.3.1 目前服务注册存在的问题

在本书上册的第 4 章（微服务注册与发现）中，我们编写了一个 ServiceRegistry 的组件，将其封装在一个服务注册框架中，当应用启动时可收集服务外部配置（包括服务名称、IP 地址、端口号等），并将这些配置注册到 ZooKeeper 中，我们要做的只是在 pom.xml 配置文件中，引入该服务注册框架的 Maven 依赖即可。

当时我们觉得该方案已经非常简单易行了，但实际上却存在明显的弊端。

- （1）服务注册框架与应用程序相耦合。
- （2）容器内部应用无法获取容器外部端口号。
- （3）ZooKeeper 中的服务节点无法自行删除。

第一点弊端是架构设计存在的问题，要解决该问题，可能需要从本质上进行改变。第二点弊端是 Docker 容器技术的限制，除非我们在启动容器时通过环境变量（使用 -e 选项）将其设置到容器内部，或者开启 host 网络方式（使用 -net=host 选项）才能解决该问题。第三点弊端虽然

可以通过脚本来解决，但明显这种做法不够优雅。

我们需要改造服务注册过程，努力将该过程做到无依赖且自动化。

在经过一段时间的技术调研，我们发现了一个名为 **Registrator** 的开源项目，它能自动识别当前宿主机上所有的容器（该特性与 **cAdvisor** 相同），并将这些容器所对应的配置信息统一注册到 **ZooKeeper** 中，从而让服务实现“自注册”的目标。

7.3.2 使用 Registrator 实现服务自注册

Registrator 本质上是一个 **Docker** 容器，它用于监视宿主机上当前正在运行的 **Docker** 容器，并从中收集它们的配置信息（包括但不限于容器所拥有的 IP 地址与端口号），还能将这些配置信息添加到 **Consul**、**Etd**、**ZooKeeper** 等注册中心。

假设我们当前运行 **Docker** 容器的宿主机 IP 地址是 192.168.199.219，运行 **ZooKeeper** 的服务器所在的 IP 地址是 192.168.199.215。

我们要使用 **Registrator** 的第一步是启动 **ZooKeeper** 注册中心。

```
docker run -d -p 2181:2181 --name zookeeper zookeeper
```

当然，以上只是启动 **ZooKeeper** 容器最简单的一种方式，我们可根据实际情况添加更多的配置选项。

当 **ZooKeeper** 容器启动完毕后，我们就能在本地通过 **ZooKeeper** 命令行客户端进行访问。

```
docker run --rm -it zookeeper zkCli.sh -server 192.168.199.215:2181
```

当确保 **ZooKeeper** 成功连接后，我们便可启动 **Registrator** 容器。

```
docker run \
-d \
-v /var/run/docker.sock:/tmp/docker.sock \
--net=host \
--name=registrator \
gliderlabs/registrator \
-ip 192.168.199.219 \
zookeeper://192.168.199.215:2181/registry
```

由于 **Registrator** 容器需要通过宿主机上所运行的 **Docker** 引擎来访问其他新启动的 **Docker** 容器，因此需要使用 **-v** 选项将 **Docker** 引擎运行时所产生的 **/var/run/docker.sock** 文件映射到

Registrator 容器内部。此外，为了确保远程服务器上的 ZooKeeper 可以顺利访问 Registrator，我们还需使用 host 网络方式来启动 Registrator 容器。在启动 Registrator 应用程序时，需指定两个配置参数，一个是当前宿主机所在的 IP 地址，另一个是 ZooKeeper 注册中心的连接方式，此时可指定/registry 为所有服务配置的根路径。

为了在容器启动时随机映射一个可用的端口号，我们需要对 Ansible 的 Playbook 任务脚本加以调整，将启动容器时映射到宿主机上的端口号去掉，也就是说，将“8080:8080”改为“8080”。

```
- name: Starting Docker Container
  docker_container:
    name: "{{JOB_NAME}}"
    image: "{{docker_registry_address}}/{{ENV}}/{{JOB_NAME}}:{{BUILD_NUMBER}}"
    ports:
      - "8080"
```

ZooKeeper 与 Registrator 都已准备就绪，现在可再次通过 Jenkins 与 Ansible 部署 product-service 服务，并通过 ZooKeeper 客户端观察/registry 路径下子节点的变化。

```
ls /registry/product-service
[192.168.199.219:32768]
```

可见，registry 根节点下有一个名为 product-service 的“服务节点”，该服务节点下有一个名为 192.168.199.219:32768 的“地址节点”，该地址节点中还包含一系列的相关数据。

```
get /registry/product-service/192.168.199.219:32768
{"Name":"product-service","IP":"192.168.199.219","PublicPort":32768,"PrivatePort":8080,"ContainerID":"989af8212090","Tags":[],"Attrs":{}}
cZxid = 0x5
ctime = Wed Jun 28 07:13:14 GMT 2017
mZxid = 0x5
mtime = Wed Jun 28 07:13:14 GMT 2017
pZxid = 0x5
cversion = 0
dataVersion = 0
aclVersion = 0
```

- **Name:** 表示容器名称，默认由--name 选项决定，可通过-e 选项设置 SERVICE_NAME 环境变量。
- **IP:** 表示容器映射到宿主机上的 IP 地址（外部 IP），它由启动 Registrator 时传入的-ip 选项决定。

- **PublicPort**: 表示容器映射到宿主机上的端口号（外部端口），它由 Docker 引擎自动生成。
- **PrivatePort**: 表示容器内部应用所暴露的端口号（内部端口），对于 Spring Boot 应用而言默认是 8080。
- **ContainerID**: 表示容器 ID，它由 Docker 引擎自动生成。
- **Tags**: 表示容器标签，可在启动容器时使用 `-l` 选项进行设置，一般不常用。
- **Attrs**: 表示容器所拥有的相关属性，可在启动容器时使用 `-e` 选项进行设置，一般不常用。

关于 **Registrator** 更全面的使用方法，请参考它的使用文档。

Registrator 使用文档: <http://gliderlabs.github.io/registrator/latest/>。

为了便于大家更加清楚地了解微服务部署与运行这两个阶段，我们画了一张流程图，如图 7-11 所示。

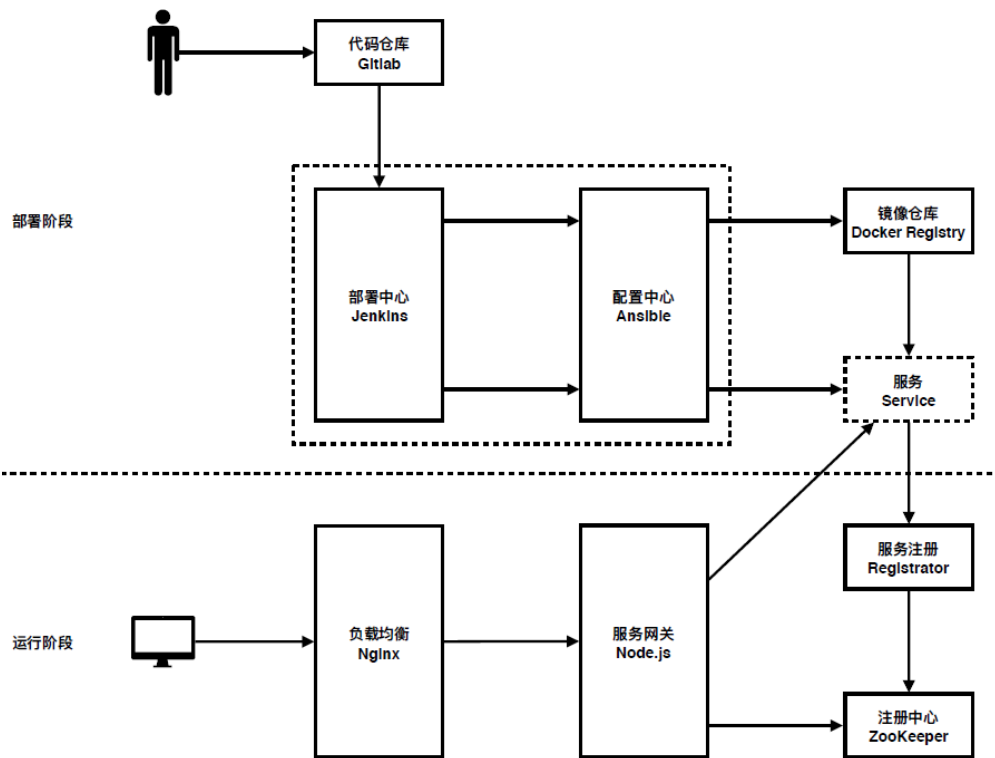


图 7-11 微服务的部署与运行阶段

在运行阶段,浏览器首先访问 Nginx 负载均衡服务器,并通过 Node.js 服务网关从 ZooKeeper 注册中心中查询到服务具体的服务配置 (IP 地址与端口号),并使用该服务配置以反向代理的方式调用具体的服务。

可见,如果当前服务正在运行中,此时就不应该进行部署,否则可能将会对用户正常操作造成影响。如果希望避免这个影响,那么必须确保微服务的升级的平滑性。

7.3.3 微服务平滑升级解决方案

我们可利用“版本号”技术轻松解决微服务平滑升级问题,具体思路如下所述。

假设当前有一个微服务,名称为 foo,当前版本号为 1 (版本号可从 Jenkins 的 BUILD_NUMBER 中获取)。当该服务部署成功后,将在 ZooKeeper 中注册一条服务配置。

```
/registry/foo-1/ip1:port1 => {...}
```

可见,该服务的完整名称为 foo-1,它包括服务本身的名称与服务的版本号,我们正是通过版本号来区分同种类型的服务的。

现在我们需要在 ZooKeeper 中定义一个 version 节点,该节点用于管理所有服务的当前版本号。

```
/version/foo => 1
```

此时对于服务网关而言,它首先从 /version 中获取某服务的当前版本号,然后通过“/registry/服务名称-版本号”来获取该服务对应的服务配置,如图 7-12 所示。

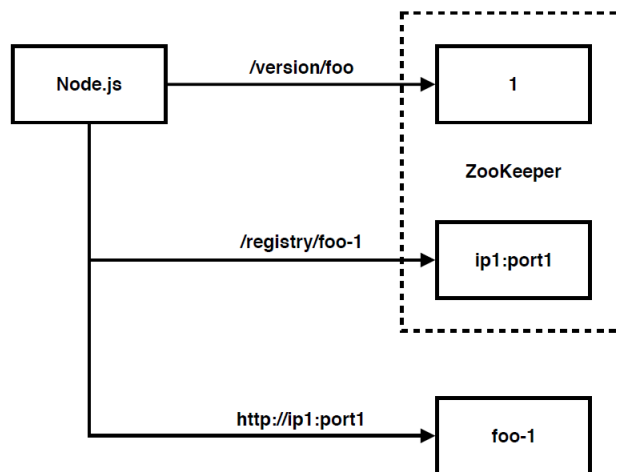


图 7-12 服务发现过程

随后我们部署一个新的服务，此时的版本号为 2，当部署成功后，在 ZooKeeper 中将看到新的服务配置。

```
/registry/foo-2/ip2:port2 => {...}
/version/foo => 2
```

为了使服务网关获取最新的服务配置 ip2:port2，我们需确保这个过程是平滑的，不会因为版本号切换而带来服务的不稳定性。

因此，我们在部署服务的过程中使用一些技巧来实现微服务的平滑升级。

为了启动 foo-2 容器，我们首先在 Ansible Playbook 任务脚本中通过 SERVICE_NAME 环境变量设置服务名称。随后通过一个 Shell 脚本在 ZooKeeper 中判断 /version/foo 节点是否已存在，若不存在，否则创建 /version/foo 节点，否则获取 /version/foo 节点中的数据（此时为 1）。最后移出 foo-1 容器，同时 Registrator 可确保 /registry/foo-1/ip1:port1 节点也将自动移出，如图 7-13 所示。

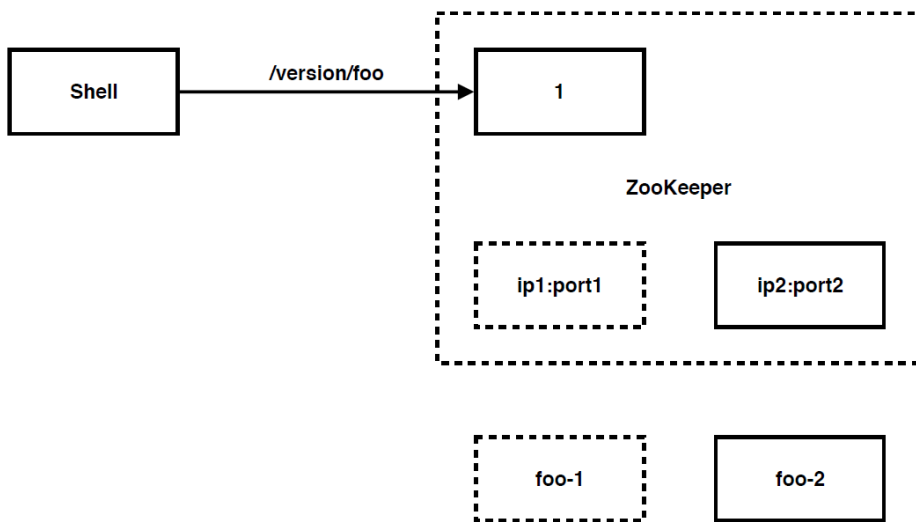


图 7-13 服务升级过程

至此 foo-2 服务部署完毕，服务发现可正常使用。

至于这个 Shell 脚本应该如何编写？整个升级过程能否做到真正平滑？在服务升级过程中是否还遗漏了哪些重要细节？

我们不妨仔细思考并亲自动手，这才是架构探险的乐趣所在。

7.4 本章小结

本章解决了微服务的配置问题，让应用程序与配置参数相分离。首先我们通过一些实例快速学习了 Ansible 自动化运维工具的使用方法，让我们深深地体会到 Ansible 可谓自动化运维之利器。随后我们使用 Ansible 作为轻量级微服务架构的“配置中心”，并结合 Jenkins 与 Ansible，优化了我们现有的微服务部署框架，通过 Ansible 不仅可管理配置参数，还能启动远程服务器中的 Docker 容器。最后我们使用 Registrator 所提供的自注册特性，实现了微服务的平滑升级目标，解决了因服务升级时导致系统面临短暂停机的问题。

虽然微服务可做到“无停机”升级，但目前还无法做到服务的高可用性，比如某个微服务因故障而宕机了，此时应该如何确保整个应用系统不会受到任何影响呢？大家不妨思考一下应该如何解决该问题。

虽然本书即将完结，但架构探险之路从未停止，我们将在后续通过在线的方式为大家传递新的探险历程，请关注“架构探险图书”公众号。

我们相信，微服务架构绝不是“昂贵”的技术，任何技术团队都应该“用得起”，它将变得更加容易落地，并始终朝着“轻量级”技术路线发展。

加油，微服务！